

Table of Contents

| | |
|-----------------|-------|
| Introduction | 1.1 |
| 准备工作 | 1.2 |
| 认识 FLow | 1.2.1 |
| Vue.js 源码目录设计 | 1.2.2 |
| Vue.js 源码构建 | 1.2.3 |
| 从入口开始 | 1.2.4 |
| 数据驱动 | 1.3 |
| new Vue 发生了什么 | 1.3.1 |
| Vue 实例挂载的实现 | 1.3.2 |
| render | 1.3.3 |
| Virtual DOM | 1.3.4 |
| createElement | 1.3.5 |
| update | 1.3.6 |
| 组件化 | 1.4 |
| createComponent | 1.4.1 |
| patch | 1.4.2 |
| 合并配置 | 1.4.3 |
| 生命周期 | 1.4.4 |
| 组件注册 | 1.4.5 |
| 异步组件 | 1.4.6 |
| 深入响应式原理 | 1.5 |
| 响应式对象 | 1.5.1 |
| 依赖收集 | 1.5.2 |
| 派发更新 | 1.5.3 |
| nextTick | 1.5.4 |
| 检测变化的注意事项 | 1.5.5 |
| 计算属性 VS 倾听属性 | 1.5.6 |
| 组件更新 | 1.5.7 |
| 原理图 | 1.5.8 |
| 编译 | 1.6 |
| 编译入口 | 1.6.1 |
| parse | 1.6.2 |
| optimize | 1.6.3 |

| | |
|------------------|-------|
| codegen | 1.6.4 |
| 扩展 | 1.7 |
| event | 1.7.1 |
| v-model | 1.7.2 |
| slot | 1.7.3 |
| keep-alive | 1.7.4 |
| transition | 1.7.5 |
| transition-group | 1.7.6 |
| Vue-Router | 1.8 |
| 路由注册 | 1.8.1 |
| VueRouter 对象 | 1.8.2 |
| matcher | 1.8.3 |
| 路径切换 | 1.8.4 |
| Vuex | 1.9 |
| Vuex 初始化 | 1.9.1 |
| API | 1.9.2 |
| 插件 | 1.9.3 |

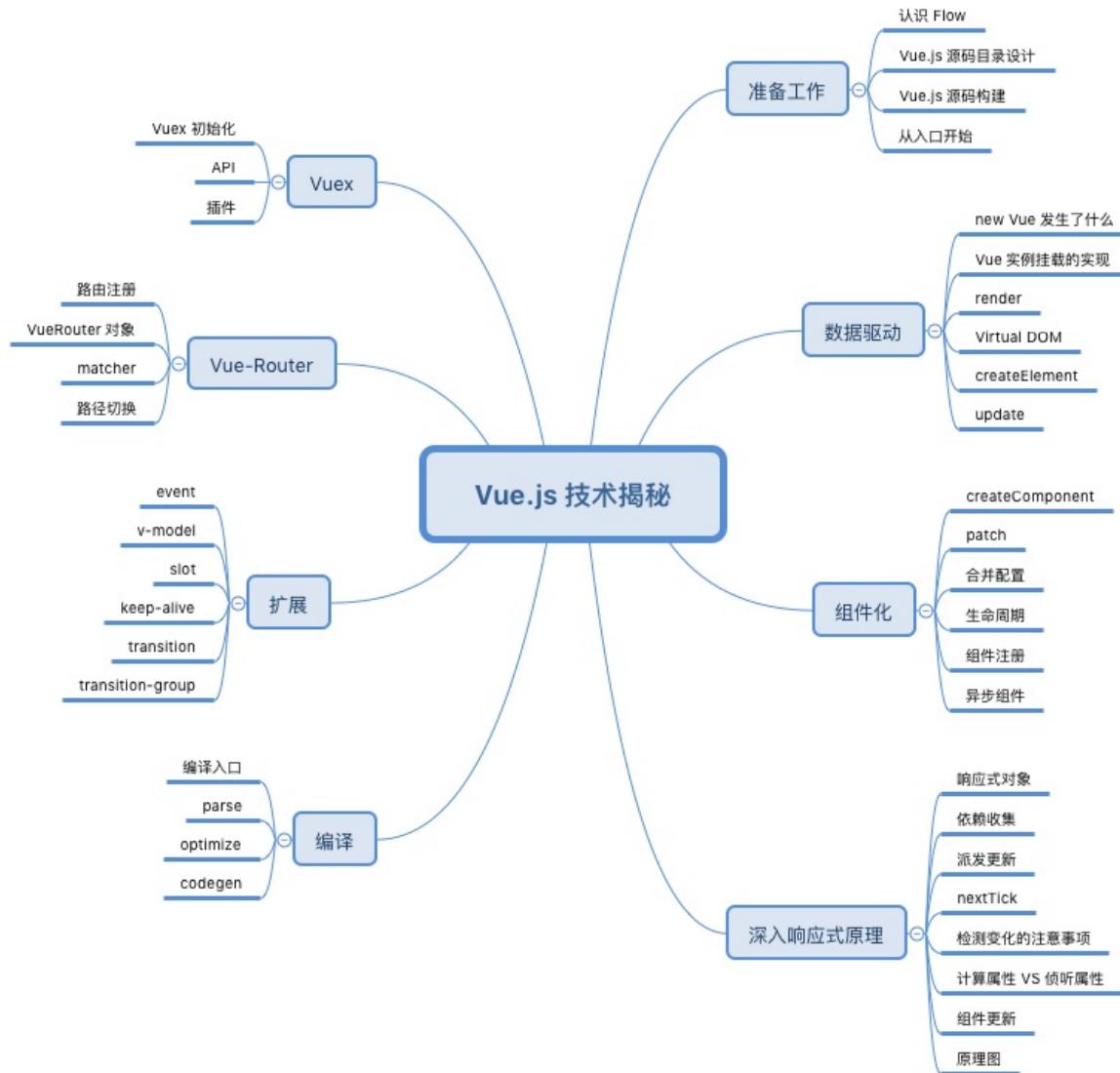
Vue.js 技术揭秘

目前社区有很多 Vue.js 的源码解析文章，但是质量层次不齐，不够系统和全面，这本电子书的目标是全方位细致深度解析 Vue.js 的实现原理，让同学们可以彻底掌握 Vue.js。目前分析的版本是 Vue.js 的最新版本 Vue.js 2.5.17-beta.0，并且之后会随着版本升级而做相应的更新，充分发挥电子书的优势。

这本电子书是作为《Vue.js 源码揭秘》视频课程的辅助教材。电子书是开源的，同学们可以免费阅读，视频是收费的，25+小时纯干货课程，如果有需要的同学可以购买来学习，但请务必支持正版，请尊重作者的劳动成果。

章节目录

为了把 Vue.js 的源码讲明白，课程设计成由浅入深，分为核心、编译、扩展、生态四个方面去讲总共，并拆成了八个章节，如下图：



第一章：准备工作

介绍了 Flow、Vue.js 的源码目录设计、Vue.js 的源码构建方式，以及从入口开始分析了 Vue.js 的初始化过程。

第二章：数据驱动

详细讲解了模板数据到 DOM 渲染的过程，从 `new Vue` 开始，分析了 `mount`、`render`、`update`、`patch` 等流程。

第三章：组件化

分析了组件化的实现原理，并且分析了组件周边的原理实现，包括合并配置、生命周期、组件注册、异步组件。

第四章：深入响应式原理

详细讲解了数据的变化如何驱动视图的变化，分析了响应式对象的创建，依赖收集、派发更新的实现过程，一些特殊情况的处理，并对比了计算属性和侦听属性的实现，最后分析了组件更新的过程。

第五章：编译

从编译的入口函数开始，分析了编译的三个核心流程的实现：`parse` -> `optimize` -> `codegen`。

第六章：扩展

详细讲解了 `event`、`v-model`、`slot`、`keep-alive`、`transition`、`transition-group` 等常用功能的原理实现，该章节作为一个可扩展章节，未来会分析更多 Vue 提供的特性。

第七章：Vue-Router

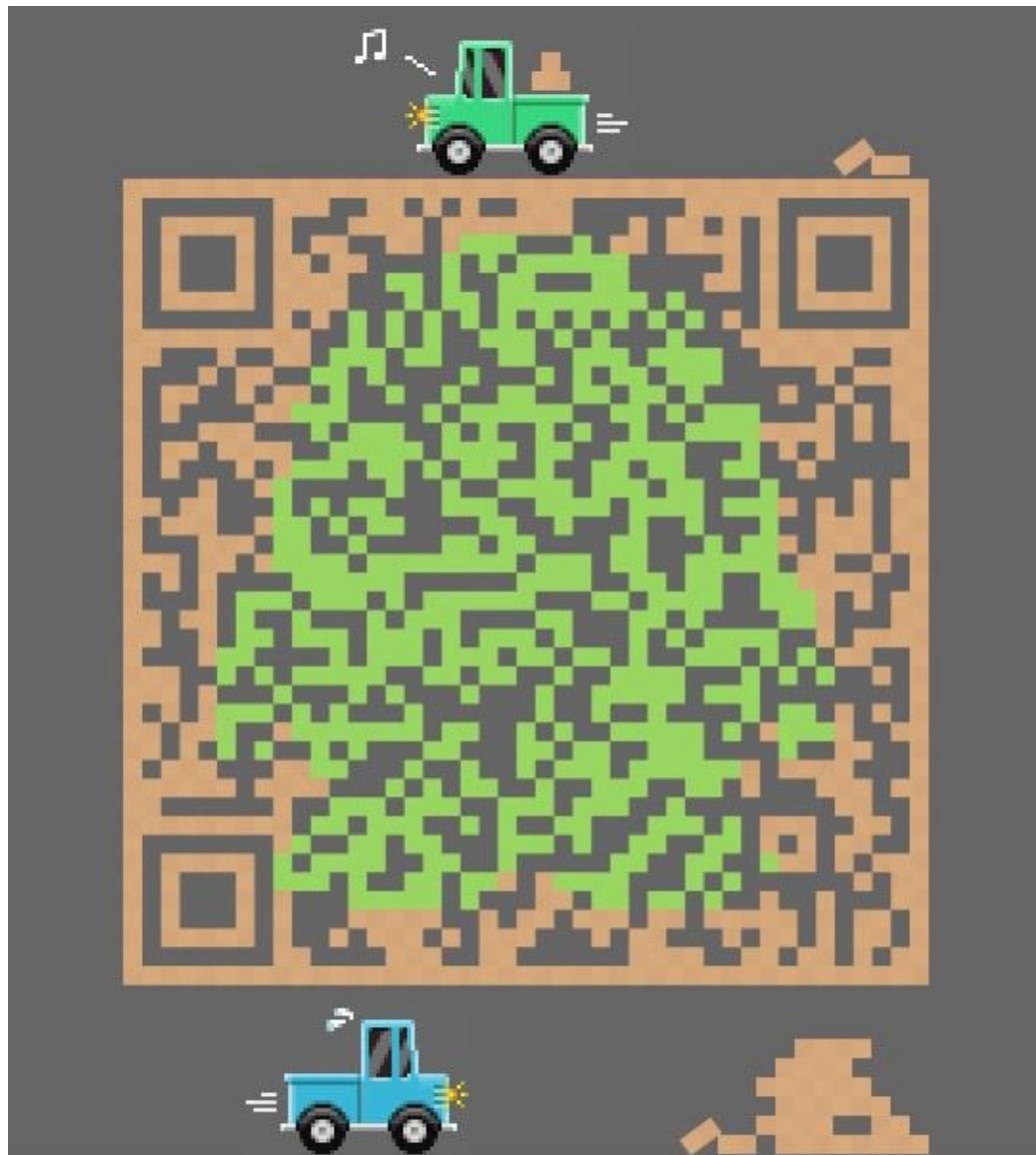
分析了 Vue-Router 的实现原理，从路由注册开始，分析了路由对象、`matcher`，并深入分析了整个路径切换的实现过程和细节。

第八章：Vuex

分析了 Vuex 的实现原理，深入分析了它的初始化过程，常用 API 以及插件部分的实现。

粉丝福利

对于通过正版渠道购买课程的同学，快来做我的粉丝吧，验证信息就是你们的订单号，感谢对正版的支持，比心~



黄老师粉丝群

扫一扫二维码，加入群聊。

准备工作

那么从这一章开始我们即将分析 Vue 的源码，我们将会介绍一些前置知识如 flow、源码目录、构建方式、编译入口等。

除此之外，我希望你已经用过 Vue 做过 2 个以上的实际项目，对 Vue 的思想有了一定的了解，对绝大部分的 API 都已经有使用。同时，我也要求你有一定的原生 JavaScript 的功底，并对代码调试有一定的了解。

如果你具备了以上条件，并且对 Vue 的实现原理很感兴趣，那么就可以开始这门课程的学习了，我将会为你打开 Vue 的底层世界大门，对它的实现细节一探究竟。

认识 Flow

Flow 是 facebook 出品的 JavaScript 静态类型检查工具。Vue.js 的源码利用了 Flow 做了静态类型检查，所以了解 Flow 有助于我们阅读源码。

为什么用 Flow

JavaScript 是动态类型语言，它的灵活性有目共睹，但是过于灵活的副作用是很容易就写出非常隐蔽的隐患代码，在编译期甚至看上去都不会报错，但在运行阶段就可能出现各种奇怪的 bug。

类型检查是当前动态类型语言的发展趋势，所谓类型检查，就是在编译期尽早发现（由类型错误引起的）bug，又不影响代码运行（不需要运行时动态检查类型），使编写 JavaScript 具有和编写 Java 等强类型语言相近的体验。

项目越复杂就越需要通过工具的手段来保证项目的维护性和增强代码的可读性。Vue.js 在做 2.0 重构的时候，在 ES2015 的基础上，除了 ESLint 保证代码风格之外，也引入了 Flow 做静态类型检查。之所以选择 Flow，主要是因为 Babel 和 ESLint 都有对应的 Flow 插件以支持语法，可以完全沿用现有的构建配置，非常小成本的改动就可以拥有静态类型检查的能力。

Flow 的工作方式

通常类型检查分成 2 种方式：

- **类型推断**：通过变量的使用上下文来推断出变量类型，然后根据这些推断来检查类型。
- **类型注释**：事先注释好我们期待的类型，Flow 会基于这些注释来判断。

类型判断

它不需要任何代码修改即可进行类型检查，最小化开发者的工作量。它不会强制你改变开发习惯，因为它会自动推断出变量的类型。这就是所谓的类型推断，Flow 最重要的特性之一。

通过一个简单例子说明一下：

```
/*@flow*/``

function split(str) {
    return str.split(' ')
}

split(11)
```

Flow 检查上述代码后会报错，因为函数 `split` 期待的参数是字符串，而我们输入了数字。

类型注释

如上所述，类型推断是 Flow 最有用的特点之一，不需要编写类型注释就能获取有用的反馈。但在某些特定的场景下，添加类型注释可以提供更好更明确的检查依据。

考虑如下代码：

```
/*@flow*/  
  
function add(x, y){  
    return x + y  
}  
  
add('Hello', 11)
```

Flow 检查上述代码时检查不出任何错误，因为从语法层面考虑，`+` 即可以用在字符串上，也可以用在数字上，我们并没有明确指出 `add()` 的参数必须为数字。

在这种情况下，我们可以借助类型注释来指明期望的类型。类型注释是以冒号 `:` 开头，可以在函数参数，返回值，变量声明中使用。

如果我们在上段代码中添加类型注释，就会变成如下：

```
/*@flow*/  
  
function add(x: number, y: number): number {  
    return x + y  
}  
  
add('Hello', 11)
```

现在 Flow 就能检查出错误，因为函数参数的期待类型为数字，而我们提供了字符串。

上面的例子是针对函数的类型注释。接下来我们来看看 Flow 能支持的一些常见的类型注释。

数组

```
/*@flow*/  
  
var arr: Array<number> = [1, 2, 3]  
  
arr.push('Hello')
```

数组类型注释的格式是 `Array<T>`，`T` 表示数组中每项的数据类型。在上述代码中，`arr` 是每项均为数字的数组。如果我们给这个数组添加了一个字符串，Flow 能检查出错误。

类和对象

```
/*@flow*/
```

```

class Bar {
  x: string;           // x 是字符串
  y: string | number; // y 可以是字符串或者数字
  z: boolean;

  constructor(x: string, y: string | number) {
    this.x = x
    this.y = y
    this.z = false
  }
}

var bar: Bar = new Bar('hello', 4)

var obj: { a: string, b: number, c: Array<string>, d: Bar } = {
  a: 'hello',
  b: 11,
  c: ['hello', 'world'],
  d: new Bar('hello', 3)
}

```

类的类型注释格式如上，可以对类自身的属性做类型检查，也可以对构造函数的参数做类型检查。这里需要注意的是，属性 `y` 的类型中间用 `|` 做间隔，表示 `y` 的类型即可以是字符串也可以是数字。

对象的注释类型类似于类，需要指定对象属性的类型。

Null

若想任意类型 `T` 可以为 `null` 或者 `undefined`，只需类似如下写成 `?T` 的格式即可。

```

/*@flow*/
var foo: ?string = null

```

此时，`foo` 可以为字符串，也可以为 `null`。

目前我们只列举了 Flow 的一些常见的类型注释。如果想了解所有类型注释，请移步 Flow 的[官方文档](#)。

Flow 在 Vue.js 源码中的应用

有时候我们想引用第三方库，或者自定义一些类型，但 Flow 并不认识，因此检查的时候会报错。为了解决这类问题，Flow 提出了一个 `libdef` 的概念，可以用来识别这些第三方库或者是自定义类型，而 Vue.js 也利用了这一特性。

在 Vue.js 的主目录下有 `.flowconfig` 文件，它是 Flow 的配置文件，感兴趣的同学可以看[官方文档](#)。这其中的 `[libs]` 部分用来描述包含指定库定义的目录，默认是名为 `flow-typed` 的目录。

这里 `[libs]` 配置的是 `flow`，表示指定的库定义都在 `flow` 文件夹内。我们打开这个目录，会发现文件如下：

```
flow
├── compiler.js      # 编译相关
├── component.js     # 组件数据结构
├── global-api.js    # Global API 结构
├── modules.js        # 第三方库定义
├── options.js        # 选项相关
├── ssr.js            # 服务端渲染相关
└── vnode.js          # 虚拟 node 相关
```

可以看到，`Vue.js` 有很多自定义类型的定义，在阅读源码的时候，如果遇到某个类型并想了解它完整的数据结构的时候，可以回来翻阅这些数据结构的定义。

总结

通过对 Flow 的认识，有助于我们阅读 Vue 的源码，并且这种静态类型检查的方式非常有利于大型项目源码的开发和维护。类似 Flow 的工具还有如 TypeScript，感兴趣的同学也可以自行去了解一下。

Vue.js 源码目录设计

Vue.js 的源码都在 src 目录下，其目录结构如下。

```

src
├── compiler      # 编译相关
├── core          # 核心代码
├── platforms     # 不同平台的支持
├── server         # 服务端渲染
├── sfc            # .vue 文件解析
└── shared         # 共享代码

```

compiler

compiler 目录包含 Vue.js 所有编译相关的代码。它包括把模板解析成 ast 语法树，ast 语法树优化，代码生成等功能。

编译的工作可以在构建时做（借助 webpack、vue-loader 等辅助插件）；也可以在运行时做，使用包含构建功能的 Vue.js。显然，编译是一项耗性能的工作，所以更推荐前者——离线编译。

core

core 目录包含了 Vue.js 的核心代码，包括内置组件、全局 API 封装，Vue 实例化、观察者、虚拟 DOM、工具函数等等。

这里的代码可谓是 Vue.js 的灵魂，也是我们之后需要重点分析的地方。

platform

Vue.js 是一个跨平台的 MVVM 框架，它可以跑在 web 上，也可以配合 weex 跑在 natvie 客户端上。platform 是 Vue.js 的入口，2 个目录代表 2 个主要入口，分别打包成运行在 web 上和 weex 上的 Vue.js。

我们会重点分析 web 入口打包后的 Vue.js，对于 weex 入口打包的 Vue.js，感兴趣的同学可以自行研究。

server

Vue.js 2.0 支持了服务端渲染，所有服务端渲染相关的逻辑都在这个目录下。注意：这部分代码是跑在服务端的 Node.js，不要和跑在浏览器端的 Vue.js 混为一谈。

服务端渲染主要的工作是把组件渲染为服务器端的 HTML 字符串，将它们直接发送到浏览器，最后将静态标记“混合”为客户端上完全交互的应用程序。

sfc

通常我们开发 Vue.js 都会借助 webpack 构建， 然后通过 .vue 单文件的编写组件。

这个目录下的代码逻辑会把 .vue 文件内容解析成一个 JavaScript 的对象。

shared

Vue.js 会定义一些工具方法，这里定义的工具方法都是会被浏览器端的 Vue.js 和服务端的 Vue.js 所共享的。

总结

从 Vue.js 的目录设计可以看到，作者把功能模块拆分的非常清楚，相关的逻辑放在一个独立的目录下维护，并且把复用的代码也抽成一个独立目录。

这样的目录设计让代码的阅读性和可维护性都变强，是非常值得学习和推敲的。

Vue.js 源码构建

Vue.js 源码是基于 [Rollup](#) 构建的，它的构建相关配置都在 scripts 目录下。

构建脚本

通常一个基于 NPM 托管的项目都会有一个 package.json 文件，它是对项目的描述文件，它的内容实际上是一个标准的 JSON 对象。

我们通常会配置 `script` 字段作为 NPM 的执行脚本，Vue.js 源码构建的脚本如下：

```
{
  "script": {
    "build": "node scripts/build.js",
    "build:ssr": "npm run build -- web-runtime-cjs,web-server-renderer",
    "build:weex": "npm run build --weex"
  }
}
```

这里总共有 3 条命令，作用都是构建 Vue.js，后面 2 条是在第一条命令的基础上，添加一些环境参数。

当在命令行运行 `npm run build` 的时候，实际上就会执行 `node scripts/build.js`，接下来我们来看看它实际是怎么构建的。

构建过程

我们对于构建过程分析是基于源码的，先打开构建的入口 JS 文件，在 `scripts/build.js` 中：

```
let builds = require('./config').getAllBuilds()

// filter builds via command line arg
if (process.argv[2]) {
  const filters = process.argv[2].split(',')
  builds = builds.filter(b => {
    return filters.some(f => b.output.file.indexOf(f) > -1 || b._name.indexOf(f) >
-1)
  })
} else {
  // filter out weex builds by default
  builds = builds.filter(b => {
    return b.output.file.indexOf('weex') === -1
  })
}

build(builds)
```

这段代码逻辑非常简单，先从配置文件读取配置，再通过命令行参数对构建配置做过滤，这样就可以构建出不同用途的 Vue.js 了。接下来我们看一下配置文件，在 `scripts/config.js` 中：

```
const builds = {
  // Runtime only (CommonJS). Used by bundlers e.g. Webpack & Browserify
  'web-runtime-cjs': {
    entry: resolve('web/entry-runtime.js'),
    dest: resolve('dist/vue.runtime.common.js'),
    format: 'cjs',
    banner
  },
  // Runtime+compiler CommonJS build (CommonJS)
  'web-full-cjs': {
    entry: resolve('web/entry-runtime-with-compiler.js'),
    dest: resolve('dist/vue.common.js'),
    format: 'cjs',
    alias: { he: './entity-decoder' },
    banner
  },
  // Runtime only (ES Modules). Used by bundlers that support ES Modules,
  // e.g. Rollup & Webpack 2
  'web-runtime-esm': {
    entry: resolve('web/entry-runtime.js'),
    dest: resolve('dist/vue.runtime.esm.js'),
    format: 'es',
    banner
  },
  // Runtime+compiler CommonJS build (ES Modules)
  'web-full-esm': {
    entry: resolve('web/entry-runtime-with-compiler.js'),
    dest: resolve('dist/vue.esm.js'),
    format: 'es',
    alias: { he: './entity-decoder' },
    banner
  },
  // runtime-only build (Browser)
  'web-runtime-dev': {
    entry: resolve('web/entry-runtime.js'),
    dest: resolve('dist/vue.runtime.js'),
    format: 'umd',
    env: 'development',
    banner
  },
  // runtime-only production build (Browser)
  'web-runtime-prod': {
    entry: resolve('web/entry-runtime.js'),
    dest: resolve('dist/vue.runtime.min.js'),
    format: 'umd',
    env: 'production',
    banner
  }
}
```

```

},
// Runtime+compiler development build (Browser)
'web-full-dev': {
  entry: resolve('web/entry-runtime-with-compiler.js'),
  dest: resolve('dist/vue.js'),
  format: 'umd',
  env: 'development',
  alias: { he: './entity-decoder' },
  banner
},
// Runtime+compiler production build (Browser)
'web-full-prod': {
  entry: resolve('web/entry-runtime-with-compiler.js'),
  dest: resolve('dist/vue.min.js'),
  format: 'umd',
  env: 'production',
  alias: { he: './entity-decoder' },
  banner
},
// ...
}
```

```

这里列举了一些 Vue.js 构建的配置，关于还有一些服务端渲染 webpack 插件以及 weex 的打包配置就不列举了。

对于单个配置，它是遵循 Rollup 的构建规则的。其中 `entry` 属性表示构建的入口 JS 文件地址，`dest` 属性表示构建后的 JS 文件地址。`format` 属性表示构建的格式，`cjs` 表示构建出来的文件遵循 CommonJS 规范，`es` 表示构建出来的文件遵循 ES Module 规范。`umd` 表示构建出来的文件遵循 UMD 规范。

以 `web-runtime-cjs` 配置为例，它的 `entry` 是 `resolve('web/entry-runtime.js')`，先来看一下 `resolve` 函数的定义。

源码目录： scripts/config.js

```

const aliases = require('./alias')
const resolve = p => {
 const base = p.split('/')[0]
 if (aliases[base]) {
 return path.resolve(aliases[base], p.slice(base.length + 1))
 } else {
 return path.resolve(__dirname, '../', p)
 }
}
```

```

这里的 `resolve` 函数实现非常简单，它先把 `resolve` 函数传入的参数 `p` 通过 `/` 做了分割成数组，然后取数组第一个元素设置为 `base`。在我们这个例子中，参数 `p` 是 `web/entry-runtime.js`，那么 `base` 则为 `web`。`base` 并不是实际的路径，它的真实路径借助了别名的配

置，我们来看一下别名配置的代码，在 `scripts/alias` 中：

```
const path = require('path')

module.exports = {
  vue: path.resolve(__dirname, '../src/platforms/web/entry-runtime-with-compiler'),
  compiler: path.resolve(__dirname, '../src/compiler'),
  core: path.resolve(__dirname, '../src/core'),
  shared: path.resolve(__dirname, '../src/shared'),
  web: path.resolve(__dirname, '../src/platforms/web'),
  weex: path.resolve(__dirname, '../src/platforms/weex'),
  server: path.resolve(__dirname, '../src/server'),
  entries: path.resolve(__dirname, '../src/entries'),
  sfc: path.resolve(__dirname, '../src/sfc')
}
```

很显然，这里 `web` 对应的真实的路径是 `path.resolve(__dirname, '../src/platforms/web')`，这个路径就找到了 Vue.js 源码的 `web` 目录。然后 `resolve` 函数通过 `path.resolve(aliases[base], p.slice(base.length + 1))` 找到了最终路径，它就是 Vue.js 源码 `web` 目录下的 `entry-runtime.js`。因此，`web-runtime-cjs` 配置对应的入口文件就找到了。

它经过 Rollup 的构建打包后，最终会在 `dist` 目录下生成 `vue.runtime.common.js`。

Runtime Only VS Runtime+Compiler

通常我们利用 `vue-cli` 去初始化我们的 Vue.js 项目的时候会询问我们用 Runtime Only 版本的还是 Runtime+Compiler 版本。下面我们来对比这两个版本。

- Runtime Only

我们在使用 Runtime Only 版本的 Vue.js 的时候，通常需要借助如 `webpack` 的 `vue-loader` 工具把 `.vue` 文件编译成 `JavaScript`，因为是在编译阶段做的，所以它只包含运行时的 `Vue.js` 代码，因此代码体积也会更轻量。

- Runtime+Compiler

我们如果没有对代码做预编译，但又使用了 `Vue` 的 `template` 属性并传入一个字符串，则需要在客户端编译模板，如下所示：

```
// 需要编译器的版本
new Vue({
  template: '<div>{{ hi }}</div>'
})

// 这种情况不需要
new Vue({
  render (h) {
    return h('div', this.hi)
  }
})
```

```
})
```

因为在 Vue.js 2.0 中，最终渲染都是通过 `render` 函数，如果写 `template` 属性，则需要编译成 `render` 函数，那么这个编译过程会发生运行时，所以需要带有编译器的版本。

很显然，这个编译过程对性能会有一定损耗，所以通常我们更推荐使用 Runtime-Only 的 Vue.js。

总结

通过这一节的分析，我们可以了解到 Vue.js 的构建打包过程，也知道了不同作用和功能的 Vue.js 它们对应的入口以及最终编译生成的 JS 文件。尽管在实际开发过程中我们会用 Runtime Only 版本开发比较多，但为了分析 Vue 的编译过程，我们这门课重点分析的源码是 Runtime+Compiler 的 Vue.js。

从入口开始

我们之前提到过 Vue.js 构建过程，在 web 应用下，我们来分析 Runtime + Compiler 构建出来的 Vue.js，它的入口是 `src/platforms/web/entry-runtime-with-compiler.js`：

```
/* @flow */

import config from 'core/config'
import { warn, cached } from 'core/util/index'
import { mark, measure } from 'core/util/perf'

import Vue from './runtime/index'
import { query } from './util/index'
import { compileToFunctions } from './compiler/index'
import { shouldDecodeNewlines, shouldDecodeNewlinesForHref } from './util/compat'

const idToTemplate = cached(id => {
  const el = query(id)
  return el && el.innerHTML
})

const mount = Vue.prototype.$mount
Vue.prototype.$mount = function (
  el?: string | Element,
  hydrating?: boolean
): Component {
  el = el && query(el)

  /* istanbul ignore if */
  if (el === document.body || el === document.documentElement) {
    process.env.NODE_ENV !== 'production' && warn(
      `Do not mount Vue to <html> or <body> - mount to normal elements instead.`
    )
    return this
  }

  const options = this.$options
  // resolve template/el and convert to render function
  if (!options.render) {
    let template = options.template
    if (template) {
      if (typeof template === 'string') {
        if (template.charAt(0) === '#') {
          template = idToTemplate(template)
          /* istanbul ignore if */
          if (process.env.NODE_ENV !== 'production' && !template) {
            warn(
              `Template element not found or is empty: ${options.template}`,
            )
          }
        }
      }
    }
  }
}
```

```

        this
    )
}
}
} else if (template.nodeType) {
    template = template.innerHTML
} else {
    if (process.env.NODE_ENV !== 'production') {
        warn('invalid template option:' + template, this)
    }
    return this
}
} else if (el) {
    template = getOuterHTML(el)
}
if (template) {
    /* istanbul ignore if */
    if (process.env.NODE_ENV !== 'production' && config.performance && mark) {
        mark('compile')
    }

    const { render, staticRenderFns } = compileToFunctions(template, {
        shouldDecodeNewlines,
        shouldDecodeNewlinesForHref,
        delimiters: options.delimiters,
        comments: options.comments
    }, this)
    options.render = render
    options.staticRenderFns = staticRenderFns

    /* istanbul ignore if */
    if (process.env.NODE_ENV !== 'production' && config.performance && mark) {
        mark('compile end')
        measure(`vue ${this._name} compile`, 'compile', 'compile end')
    }
}
}
return mount.call(this, el, hydrating)
}

/**
 * Get outerHTML of elements, taking care
 * of SVG elements in IE as well.
 */
function getOuterHTML (el: Element): string {
    if (el.outerHTML) {
        return el.outerHTML
    } else {
        const container = document.createElement('div')
        container.appendChild(el.cloneNode(true))
        return container.innerHTML
    }
}

```

```

        }
    }

Vue.compile = compileToFunctions

export default Vue

```

那么，当我们的代码执行 `import Vue from 'vue'` 的时候，就是从这个入口执行代码来初始化 Vue，那么 Vue 到底是什么，它是怎么初始化的，我们来一探究竟。

Vue 的入口

在这个入口 JS 的上方我们可以找到 `Vue` 的来源：`import Vue from './runtime/index'`，我们先来看一下这块儿的实现，它定义在 `src/platforms/web/runtime/index.js` 中：

```

import Vue from 'core/index'
import config from 'core/config'
import { extend, noop } from 'shared/util'
import { mountComponent } from 'core/instance/lifecycle'
import { devtools, inBrowser, isChrome } from 'core/util/index'

import {
    query,
    mustUseProp,
    isReservedTag,
    isReservedAttr,
    getTagNameSpace,
    isUnknownElement
} from 'web/util/index'

import { patch } from './patch'
import platformDirectives from './directives/index'
import platformComponents from './components/index'

// install platform specific utils
Vue.config.mustUseProp = mustUseProp
Vue.config.isReservedTag = isReservedTag
Vue.config.isReservedAttr = isReservedAttr
Vue.config.getTagNameSpace = getTagNameSpace
Vue.config.isUnknownElement = isUnknownElement

// install platform runtime directives & components
extend(Vue.options.directives, platformDirectives)
extend(Vue.options.components, platformComponents)

// install platform patch function
Vue.prototype.__patch__ = inBrowser ? patch : noop

// public mount method

```

```

Vue.prototype.$mount = function (
  el?: string | Element,
  hydrating?: boolean
): Component {
  el = el && inBrowser ? query(el) : undefined
  return mountComponent(this, el, hydrating)
}

// ...

export default Vue

```

这里关键的代码是 `import Vue from 'core/index'`，之后的逻辑都是对 Vue 这个对象做一些扩展，可以先不用看，我们来看一下真正初始化 Vue 的地方，在 `src/core/index.js` 中：

```

import Vue from './instance/index'
import { initGlobalAPI } from './global-api/index'
import { isServerRendering } from 'core/util/env'
import { FunctionalRenderContext } from 'core/vdom/create-functional-component'

initGlobalAPI(Vue)

Object.defineProperty(Vue.prototype, '$isServer', {
  get: isServerRendering
})

Object.defineProperty(Vue.prototype, '$ssrContext', {
  get () {
    /* istanbul ignore next */
    return this.$vnode && this.$vnode.ssrContext
  }
})

// expose FunctionalRenderContext for ssr runtime helper installation
Object.defineProperty(Vue, 'FunctionalRenderContext', {
  value: FunctionalRenderContext
})

Vue.version = '__VERSION__'

export default Vue

```

这里有 2 处关键的代码，`import Vue from './instance/index'` 和 `initGlobalAPI(Vue)`，初始化全局 Vue API（我们稍后介绍），我们先来看第一部分，在 `src/core/instance/index.js` 中：

Vue 的定义

```

import { initMixin } from './init'

```

```

import { stateMixin } from './state'
import { renderMixin } from './render'
import { eventsMixin } from './events'
import { lifecycleMixin } from './lifecycle'
import { warn } from '../util/index'

function Vue (options) {
  if (process.env.NODE_ENV !== 'production' &&
    !(this instanceof Vue)
  ) {
    warn('Vue is a constructor and should be called with the `new` keyword')
  }
  this._init(options)
}

initMixin(Vue)
stateMixin(Vue)
eventsMixin(Vue)
lifecycleMixin(Vue)
renderMixin(Vue)

export default Vue

```

在这里，我们终于看到了 Vue 的庐山真面目，它实际上就是一个用 Function 实现的类，我们只能通过 `new Vue` 去实例化它。

有些同学看到这不禁想问，为何 Vue 不用 ES6 的 Class 去实现呢？我们往后看这里有很多 `xxxMixin` 的函数调用，并把 `vue` 当参数传入，它们的功能都是给 Vue 的 prototype 上扩展一些方法（这里具体的细节会在之后的文章介绍，这里不展开），Vue 按功能把这些扩展分散到多个模块中去实现，而不是在一个模块里实现所有，这种方式是用 Class 难以实现的。这么做的好处是非常方便代码的维护和管理，这种编程技巧也非常值得我们去学习。

initGlobalAPI

Vue.js 在整个初始化过程中，除了给它的原型 prototype 上扩展方法，还会给 `vue` 这个对象本身扩展全局的静态方法，它的定义在 `src/core/global-api/index.js` 中：

```

export function initGlobalAPI (Vue: GlobalAPI) {
  // config
  const configDef = {}
  configDef.get = () => config
  if (process.env.NODE_ENV !== 'production') {
    configDef.set = () => {
      warn(
        'Do not replace the Vue.config object, set individual fields instead.'
      )
    }
  }
  Object.defineProperty(Vue, 'config', configDef)
}

```

```

// exposed util methods.
// NOTE: these are not considered part of the public API - avoid relying on
// them unless you are aware of the risk.
Vue.util = {
  warn,
  extend,
  mergeOptions,
  defineReactive
}

Vue.set = set
Vue.delete = del
Vue.nextTick = nextTick

Vue.options = Object.create(null)
ASSET_TYPES.forEach(type => {
  Vue.options[type + 's'] = Object.create(null)
})

// this is used to identify the "base" constructor to extend all plain-object
// components with in Weex's multi-instance scenarios.
Vue.options._base = Vue

extend(Vue.options.components, builtInComponents)

initUse(Vue)
initMixin(Vue)
initExtend(Vue)
initAssetRegisters(Vue)
}

```

这里就是在 `Vue` 上扩展的一些全局方法的定义，`Vue` 官网上关于全局 API 都可以在这里找到，这里不会介绍细节，会在之后的章节我们具体介绍到某个 API 的时候会详细介绍。有一点要注意的是，`Vue.util` 暴露的方法最好不要依赖，因为它可能经常会发生变化，是不稳定的。

总结

那么至此，`Vue` 的初始化过程基本介绍完毕。这一节的目的是让同学们对 `Vue` 是什么有一个直观的认识，它本质上就是一个用 `Function` 实现的 `Class`，然后它的原型 `prototype` 以及它本身都扩展了一系列的方法和属性，那么 `Vue` 能做什么，它是怎么做的，我们会在后面的章节一层层帮大家揭开 `Vue` 的神秘面纱。

数据驱动

Vue.js 一个核心思想是数据驱动。所谓数据驱动，是指视图是由数据驱动生成的，我们对视图的修改，不会直接操作 DOM，而是通过修改数据。它相比我们传统的前端开发，如使用 jQuery 等前端库直接修改 DOM，大大简化了代码量。特别是当交互复杂的时候，只关心数据的修改会让代码的逻辑变的非常清晰，因为 DOM 变成了数据的映射，我们所有的逻辑都是对数据的修改，而不用碰触 DOM，这样的代码非常利于维护。

在 Vue.js 中我们可以采用简洁的模板语法来声明式的将数据渲染为 DOM：

```
<div id="app">
  {{ message }}
</div>
```

```
var app = new Vue({
  el: '#app',
  data: {
    message: 'Hello Vue!'
  }
})
```

最终它会在页面上渲染出 `Hello Vue`。接下来，我们会从源码角度来分析 Vue 是如何实现的，分析过程会以主线代码为主，重要的分支逻辑会放在之后单独分析。数据驱动还有一部分是数据更新驱动视图变化，这一块内容我们也会在之后的章节分析，这一章我们的目标是弄清楚模板和数据如何渲染成最终的 DOM。

new Vue 发生了什么

从入口代码开始分析，我们先来分析 `new Vue` 背后发生了哪些事情。我们都知道，`new` 关键字在 Javascript 语言中代表实例化是一个对象，而 `Vue` 实际上是一个类，类在 Javascript 中是用 `Function` 来实现的，来看一下源码，在 `src/core/instance/index.js` 中。

```
function Vue (options) {
  if (process.env.NODE_ENV !== 'production' &&
    !(this instanceof Vue)
  ) {
    warn('Vue is a constructor and should be called with the `new` keyword')
  }
  this._init(options)
}
```

可以看到 `Vue` 只能通过 `new` 关键字初始化，然后会调用 `this._init` 方法，该方法在 `src/core/instance/init.js` 中定义。

```
Vue.prototype._init = function (options?: Object) {
  const vm: Component = this
  // a uid
  vm._uid = uid++

  let startTag, endTag
  /* istanbul ignore if */
  if (process.env.NODE_ENV !== 'production' && config.performance && mark) {
    startTag = `vue-perf-start:${vm._uid}`
    endTag = `vue-perf-end:${vm._uid}`
    mark(startTag)
  }

  // a flag to avoid this being observed
  vm._isVue = true
  // merge options
  if (options && options._isComponent) {
    // optimize internal component instantiation
    // since dynamic options merging is pretty slow, and none of the
    // internal component options needs special treatment.
    initInternalComponent(vm, options)
  } else {
    vm.$options = mergeOptions(
      resolveConstructorOptions(vm.constructor),
      options || {},
      vm
    )
  }
  /* istanbul ignore else */
}
```

```

if (process.env.NODE_ENV !== 'production') {
  initProxy(vm)
} else {
  vm._renderProxy = vm
}
// expose real self
vm._self = vm
initLifecycle(vm)
initEvents(vm)
initRender(vm)
callHook(vm, 'beforeCreate')
initInjections(vm) // resolve injections before data/props
initState(vm)
initProvide(vm) // resolve provide after data/props
callHook(vm, 'created')

/* istanbul ignore if */
if (process.env.NODE_ENV !== 'production' && config.performance && mark) {
  vm._name = formatComponentName(vm, false)
  mark(endTag)
  measure(`vue ${vm._name} init`, startTag, endTag)
}

if (vm.$options.el) {
  vm.$mount(vm.$options.el)
}
}

```

Vue 初始化主要就干了几件事情，合并配置，初始化生命周期，初始化事件中心，初始化渲染，初始化 data、props、computed、watcher 等等。

总结

Vue 的初始化逻辑写得非常清楚，把不同的功能逻辑拆成一些单独的函数执行，让主线逻辑一目了然，这样的编程思想是非常值得借鉴和学习的。

由于我们这一章的目标是弄清楚模板和数据如何渲染成最终的 DOM，所以各种初始化逻辑我们先不看。在初始化的最后，检测到如果有 `el` 属性，则调用 `vm.$mount` 方法挂载 `vm`，挂载的目标就是把模板渲染成最终的 DOM，那么接下来我们来分析 Vue 的挂载过程。

Vue 实例挂载的实现

Vue 中我们是通过 `$mount` 实例方法去挂载 `vm` 的，`$mount` 方法在多个文件中都有定义，如 `src/platform/web/entry-runtime-with-compiler.js`、`src/platform/web/runtime/index.js`、`src/platform/weex/runtime/index.js`。因为 `$mount` 这个方法的实现是和平台、构建方式都相关的。接下来我们重点分析带 `compiler` 版本的 `$monut` 实现，因为抛开 webpack 的 vue-loader，我们在纯前端浏览器环境分析 Vue 的工作原理，有助于我们对原理理解的深入。

`compiler` 版本的 `$monut` 实现非常有意思，先来看一下 `src/platform/web/entry-runtime-with-compiler.js` 文件中定义：

```
const mount = Vue.prototype.$mount
Vue.prototype.$mount = function (
  el?: string | Element,
  hydrating?: boolean
): Component {
  el = el && query(el)

  /* istanbul ignore if */
  if (el === document.body || el === document.documentElement) {
    process.env.NODE_ENV !== 'production' && warn(
      `Do not mount Vue to <html> or <body> - mount to normal elements instead.`
    )
    return this
  }

  const options = this.$options
  // resolve template/el and convert to render function
  if (!options.render) {
    let template = options.template
    if (template) {
      if (typeof template === 'string') {
        if (template.charAt(0) === '#') {
          template = idToTemplate(template)
        /* istanbul ignore if */
        if (process.env.NODE_ENV !== 'production' && !template) {
          warn(
            `Template element not found or is empty: ${options.template}`,
            this
          )
        }
      }
    }
  } else if (template.nodeType) {
    template = template.innerHTML
  } else {
    if (process.env.NODE_ENV !== 'production') {
      warn(`invalid template option: ${template}`, this)
    }
  }
}
```

```

        }
        return this
    }
} else if (el) {
    template = getOuterHTML(el)
}
if (template) {
    /* istanbul ignore if */
    if (process.env.NODE_ENV !== 'production' && config.performance && mark) {
        mark('compile')
    }

    const { render, staticRenderFns } = compileToFunctions(template, {
        shouldDecodeNewlines,
        shouldDecodeNewlinesForHref,
        delimiters: options.delimiters,
        comments: options.comments
    }, this)
    options.render = render
    options.staticRenderFns = staticRenderFns

    /* istanbul ignore if */
    if (process.env.NODE_ENV !== 'production' && config.performance && mark) {
        mark('compile end')
        measure(`vue ${this._name} compile`, 'compile', 'compile end')
    }
}
}
return mount.call(this, el, hydrating)
}

```

这段代码首先缓存了原型上的 `$mount` 方法，再重新定义该方法，我们先来分析这段代码。首先，它对 `el` 做了限制，Vue 不能挂载在 `body`、`html` 这样的根节点上。接下来的是很关键的逻辑——如果没有定义 `render` 方法，则会把 `el` 或者 `template` 字符串转换成 `render` 方法。这里我们要牢记，在 Vue 2.0 版本中，所有 Vue 的组件的渲染最终都需要 `render` 方法，无论我们是用单文件 `.vue` 方式开发组件，还是写了 `el` 或者 `template` 属性，最终都会转换成 `render` 方法，那么这个过程是 Vue 的一个“在线编译”的过程，它是调用 `compileToFunctions` 方法实现的，编译过程我们之后会介绍。最后，调用原先原型上的 `$mount` 方法挂载。

原先原型上的 `$mount` 方法在 `src/platform/web/runtime/index.js` 中定义，之所以这么设计完全是为了复用，因为它是可以被 `runtime only` 版本的 Vue 直接使用的。

```

// public mount method
Vue.prototype.$mount = function (
    el?: string | Element,
    hydrating?: boolean
): Component {
    el = el && inBrowser ? query(el) : undefined
    return mountComponent(this, el, hydrating)
}

```

```
}
```

`$mount` 方法支持传入 2 个参数，第一个是 `el`，它表示挂载的元素，可以是字符串，也可以是 DOM 对象，如果是字符串在浏览器环境下会调用 `query` 方法转换成 DOM 对象的。第二个参数是和服务端渲染相关，在浏览器环境下我们不需要传第二个参数。

`$mount` 方法实际上会去调用 `mountComponent` 方法，这个方法定义在 `src/core/instance/lifecycle.js` 文件中：

```
export function mountComponent (
  vm: Component,
  el: ?Element,
  hydrating?: boolean
): Component {
  vm.$el = el
  if (!vm.$options.render) {
    vm.$options.render = createEmptyVNode
    if (process.env.NODE_ENV !== 'production') {
      /* istanbul ignore if */
      if ((vm.$options.template && vm.$options.template.charAt(0) !== '#') ||
        vm.$options.el || el) {
        warn(
          'You are using the runtime-only build of Vue where the template ' +
          'compiler is not available. Either pre-compile the templates into ' +
          'render functions, or use the compiler-included build.',
          vm
        )
      } else {
        warn(
          'Failed to mount component: template or render function not defined.',
          vm
        )
      }
    }
  }
  callHook(vm, 'beforeMount')

  let updateComponent
  /* istanbul ignore if */
  if (process.env.NODE_ENV !== 'production' && config.performance && mark) {
    updateComponent = () => {
      const name = vm._name
      const id = vm._uid
      const startTag = `vue-perf-start:${id}`
      const endTag = `vue-perf-end:${id}`

      mark(startTag)
      const vnode = vm._render()
      mark(endTag)
      measure(`vue ${name} render`, startTag, endTag)
    }
  }
}
```

```

    mark(startTag)
    vm._update(vnode, hydrating)
    mark(endTag)
    measure(`vue ${name} patch`, startTag, endTag)
}
} else {
  updateComponent = () => {
    vm._update(vm._render(), hydrating)
  }
}

// we set this to vm._watcher inside the watcher's constructor
// since the watcher's initial patch may call $forceUpdate (e.g. inside child
// component's mounted hook), which relies on vm._watcher being already defined
new Watcher(vm, updateComponent, noop, {
  before () {
    if (vm._isMounted) {
      callHook(vm, 'beforeUpdate')
    }
  }
}, true /* isRenderWatcher */)

hydrating = false

// manually mounted instance, call mounted on self
// mounted is called for render-created child components in its inserted hook
if (vm.$vnode == null) {
  vm._isMounted = true
  callHook(vm, 'mounted')
}
return vm
}

```

从上面的代码可以看到，`mountComponent` 核心就是先调用 `vm._render` 方法先生成虚拟 Node，再实例化一个渲染 `Watcher`，在它的回调函数中会调用 `updateComponent` 方法，最终调用 `vm._update` 更新 DOM。

`Watcher` 在这里起到两个作用，一个是初始化的时候会执行回调函数，另一个是当 `vm` 实例中的监测的数据发生变化的时候执行回调函数，这块儿我们会在之后的章节中介绍。

函数最后判断为根节点的时候设置 `vm._isMounted` 为 `true`，表示这个实例已经挂载了，同时执行 `mounted` 钩子函数。这里注意 `vm.$vnode` 表示 Vue 实例的父虚拟 Node，所以它为 `null` 则表示当前是根 Vue 的实例。

总结

`mountComponent` 方法的逻辑也是非常清晰的，它会完成整个渲染工作，接下来我们要重点分析其中的细节，也就是最核心的 2 个方法：`vm._render` 和 `vm._update`。

render

Vue 的 `_render` 方法是实例的一个私有方法，它用来把实例渲染成一个虚拟 Node。它的定义在 `src/core/instance/render.js` 文件中：

```

Vue.prototype._render = function (): VNode {
  const vm: Component = this
  const { render, _parentVnode } = vm.$options

  // reset _rendered flag on slots for duplicate slot check
  if (process.env.NODE_ENV !== 'production') {
    for (const key in vm.$slots) {
      // $flow-disable-line
      vm.$slots[key]._rendered = false
    }
  }

  if (_parentVnode) {
    vm.$scopedSlots = _parentVnode.data.scopedSlots || emptyObject
  }

  // set parent vnode. this allows render functions to have access
  // to the data on the placeholder node.
  vm.$vnode = _parentVnode
  // render self
  let vnode
  try {
    vnode = render.call(vm._renderProxy, vm.$createElement)
  } catch (e) {
    handleError(e, vm, `render`)
    // return error render result,
    // or previous vnode to prevent render error causing blank component
    /* istanbul ignore else */
    if (process.env.NODE_ENV !== 'production') {
      if (vm.$options.renderError) {
        try {
          vnode = vm.$options.renderError.call(vm._renderProxy, vm.$createElement,
        e)
        } catch (e) {
          handleError(e, vm, `renderError`)
          vnode = vm._vnode
        }
      } else {
        vnode = vm._vnode
      }
    } else {
      vnode = vm._vnode
    }
  }
}

```

```

    }
    // return empty vnode in case the render function errored out
    if (!(vnode instanceof VNode)) {
        if (process.env.NODE_ENV !== 'production' && Array.isArray(vnode)) {
            warn(
                'Multiple root nodes returned from render function. Render function ' +
                'should return a single root node.',
                vm
            )
        }
        vnode = createEmptyVNode()
    }
    // set parent
    vnode.parent = _parentVnode
    return vnode
}

```

这段代码最关键的是 `render` 方法的调用，我们在平时的开发工作中手写 `render` 方法的场景比较少，而写的比较多的是 `template` 模板，在之前的 `mounted` 方法的实现中，会把 `template` 编译成 `render` 方法，但这个编译过程是非常复杂的，我们不打算在这里展开讲，之后会专门花一个章节来分析 Vue 的编译过程。

在 Vue 的官方文档中介绍了 `render` 函数的第一个参数是 `createElement`，那么结合之前的例子：

```

<div id="app">
  {{ message }}
</div>

```

相当于我们编写如下 `render` 函数：

```

render: function (createElement) {
    return createElement('div', {
        attrs: {
            id: 'app'
        },
        this.message
    })
}

```

再回到 `_render` 函数中的 `render` 方法的调用：

```

vnode = render.call(vm._renderProxy, vm.$createElement)

```

可以看到，`render` 函数中的 `createElement` 方法就是 `vm.$createElement` 方法：

```

export function initRender (vm: Component) {
    // ...
    // bind the createElement fn to this instance
}

```

```
// so that we get proper render context inside it.  
// args order: tag, data, children, normalizationType, alwaysNormalize  
// internal version is used by render functions compiled from templates  
vm._c = (a, b, c, d) => createElement(vm, a, b, c, d, false)  
// normalization is always applied for the public version, used in  
// user-written render functions.  
vm.$createElement = (a, b, c, d) => createElement(vm, a, b, c, d, true)  
}
```

实际上，`vm.$createElement` 方法定义是在执行 `initRender` 方法的时候，可以看到除了 `vm.$createElement` 方法，还有一个 `vm._c` 方法，它是被模板编译成的 `render` 函数使用，而 `vm.$createElement` 是用户手写 `render` 方法使用的，这两个方法支持的参数相同，并且内部都调用了 `createElement` 方法。

总结

`vm._render` 最终是通过执行 `createElement` 方法并返回的是 `vnode`，它是一个虚拟 Node。Vue 2.0 相比 Vue 1.0 最大的升级就是利用了 Virtual DOM。因此在分析 `createElement` 的实现前，我们先了解一下 Virtual DOM 的概念。

Virtual DOM

Virtual DOM 这个概念相信大部分人都不会陌生，它产生的前提是浏览器中的 DOM 是很“昂贵”的，为了更直观的感受，我们可以简单的把一个简单的 div 元素的属性都打印出来，如图所示：

```
> var div = document.createElement('div')
var str = ''
for (var key in div){
  str += key + ' '
}
<"align title lang translate dir dataset hidden tabIndex accessKey draggable spellcheck contentEditable isContentEditable
offsetParent offsetTop offsetLeft offsetWidth offsetHeight style innerText outerText webkitDropzone onabort onblur
oncancel oncanplay oncanplaythrough onchange onclick onclose oncontextmenu oncuechange ondblclick ondrag ondragend
ondragenter ondragleave ondragover ondragstart ondrop ondurationchange onemptied onended onerror onfocus oninput
oninvalid onkeydown onkeypress onkeyup onload onloadeddata onloadedmetadata onloadstart onmousedown onmouseenter
onmouseleave onmousemove onmouseout onmouseover onmouseup onmousewheel onpause onplay onplaying onprogress onratechange
onreset onresize onscroll onseeked onseeking onselect onshow onstalled onsubmit onsuspend ontimeupdate ontoggle
onwheel onwebkitfullscreenchange onwebkitfullscreenerror previousElementSibling nextElementSibling children
firstElementChild lastElementChild childElementCount hasAttributes getAttribute getAttributeNS setAttribute
setAttributeNS removeAttribute removeAttributeNS hasAttribute hasAttributeNS getAttributeNode getAttributeNodeNS
setAttributeNode setAttributeNodeNS removeAttributeNode closest matches webkitMatchesSelector getElementsByTagName
getElementsByTagNameNS getElementsByTagName insertAdjacentElement insertAdjacentText insertAdjacentHTML
createShadowRoot getDestinationInsertionPoints requestPointerLock getClientRects getBoundingClientRect scrollIntoView
scrollIntoViewIfNeeded animate remove webkitRequestFullScreen webkitRequestFullscreen querySelector querySelectorAll
ELEMENT_NODE ATTRIBUTE_NODE TEXT_NODE CDATA_SECTION_NODE ENTITY_REFERENCE_NODE ENTITY_NODE PROCESSING_INSTRUCTION_NODE
COMMENT_NODE DOCUMENT_NODE DOCUMENT_TYPE_NODE DOCUMENT_FRAGMENT_NODE NOTATION_NODE DOCUMENT_POSITION_DISCONNECTED
DOCUMENT_POSITION_PRECEDING DOCUMENT_POSITION_FOLLOWING DOCUMENT_POSITION_CONTAINS DOCUMENT_POSITION_CONTAINED_BY
DOCUMENT_POSITION_IMPLEMENTATION_SPECIFIC nodeType nodeName baseURI isConnected ownerDocument parentNode parentElement
childNodes firstChild lastChild previousSibling nextSibling nodeValue textContent hasChildNodes normalize cloneNode
isEqualNode isSameNode compareDocumentPosition contains lookupPrefix lookupNamespaceURI isDefaultNamespace insertBefore
appendChild replaceChild removeChild addEventListener removeEventListener dispatchEvent "
```

可以看到，真正的 DOM 元素是非常庞大的，因为浏览器的标准就把 DOM 设计的非常复杂。当我们频繁的去做 DOM 更新，会产生一定的性能问题。

而 Virtual DOM 就是用一个原生的 JS 对象去描述一个 DOM 节点，所以它比创建一个 DOM 的代价要小很多。在 Vue.js 中，Virtual DOM 是用 `VNode` 这么一个 Class 去描述，它是定义在 `src/core/vdom/vnode.js` 中的。

```
export default class VNode {
  tag: string | void;
  data: VNodeData | void;
  children: ?Array<VNode>;
  text: string | void;
  elm: Node | void;
  ns: string | void;
  context: Component | void; // rendered in this component's scope
  key: string | number | void;
  componentOptions: VNodeComponentOptions | void;
  componentInstance: Component | void; // component instance
  parent: VNode | void; // component placeholder node

  // strictly internal
  raw: boolean; // contains raw HTML? (server only)
  isStatic: boolean; // hoisted static node
  isRootInsert: boolean; // necessary for enter transition check
  isComment: boolean; // empty comment placeholder?
  isCloned: boolean; // is a cloned node?
  isOnce: boolean; // is a v-once node?
```

```

asyncFactory: Function | void; // async component factory function
asyncMeta: Object | void;
isAsyncPlaceholder: boolean;
ssrContext: Object | void;
fnContext: Component | void; // real context vm for functional nodes
fnOptions: ?ComponentOptions; // for SSR caching
fnScopeId: ?string; // functional scope id support

constructor (
  tag?: string,
  data?: VNodeData,
  children?: ?Array<VNode>,
  text?: string,
  elm?: Node,
  context?: Component,
  componentOptions?: VNodeComponentOptions,
  asyncFactory?: Function
) {
  this.tag = tag
  this.data = data
  this.children = children
  this.text = text
  this.elm = elm
  this.ns = undefined
  this.context = context
  this.fnContext = undefined
  this.fnOptions = undefined
  this.fnScopeId = undefined
  this.key = data && data.key
  this.componentOptions = componentOptions
  this.componentInstance = undefined
  this.parent = undefined
  this.raw = false
  this.isStatic = false
  this.isRootInsert = true
  this.isComment = false
  this.isCloned = false
  this.isOnce = false
  this.asyncFactory = asyncFactory
  this.asyncMeta = undefined
  this.isAsyncPlaceholder = false
}

// DEPRECATED: alias for componentInstance for backwards compat.
/* istanbul ignore next */
get child (): Component | void {
  return this.componentInstance
}
}

```

可以看到 Vue.js 中的 Virtual DOM 的定义还是略微复杂一些的，因为它这里包含了很多 Vue.js 的特性。这里千万不要被这些茫茫多的属性吓到，实际上 Vue.js 中 Virtual DOM 是借鉴了一个开源库 [snabbdom](#) 的实现，然后加入了一些 Vue.js 特色的东西。我建议大家如果想深入了解 Vue.js 的 Virtual DOM 前不妨先阅读这个库的源码，因为它更加简单和纯粹。

总结

其实 VNode 是对真实 DOM 的一种抽象描述，它的核心定义无非就几个关键属性，标签名、数据、子节点、键值等，其它属性都是用来扩展 VNode 的灵活性以及实现一些特殊 feature 的。由于 VNode 只是用来映射到真实 DOM 的渲染，不需要包含操作 DOM 的方法，因此它是非常轻量和简单的。

Virtual DOM 除了它的数据结构的定义，映射到真实的 DOM 实际上要经历 VNode 的 create、diff、patch 等过程。那么在 Vue.js 中，VNode 的 create 是通过之前提到的 `createElement` 方法创建的，我们接下来分析这部分的实现。

createElement

Vue.js 利用 createElement 方法创建 VNode，它定义在 `src/core/vdom/create-element.js` 中：

```
// wrapper function for providing a more flexible interface
// without getting yelled at by flow
export function createElement (
  context: Component,
  tag: any,
  data: any,
  children: any,
  normalizationType: any,
  alwaysNormalize: boolean
): VNode | Array<VNode> {
  if (Array.isArray(data) || isPrimitive(data)) {
    normalizationType = children
    children = data
    data = undefined
  }
  if (isTrue(alwaysNormalize)) {
    normalizationType = ALWAYS_NORMALIZE
  }
  return _createElement(context, tag, data, children, normalizationType)
}
```

`createElement` 方法实际上是对 `_createElement` 方法的封装，它允许传入的参数更加灵活，在处理这些参数后，调用真正创建 VNode 的函数 `_createElement`：

```
export function _createElement (
  context: Component,
  tag?: string | Class<Component> | Function | Object,
  data?: VNodeData,
  children?: any,
  normalizationType?: number
): VNode | Array<VNode> {
  if (isDef(data) && isDef((data: any).__ob__)) {
    process.env.NODE_ENV !== 'production' && warn(
      `Avoid using observed data object as vnode data: ${JSON.stringify(data)}\n` +
      'Always create fresh vnode data objects in each render!',
      context
    )
    return createEmptyVNode()
  }
  // object syntax in v-bind
  if (isDef(data) && isDef(data.is)) {
    tag = data.is
  }
```

```

if (!tag) {
  // in case of component :is set to falsy value
  return createEmptyVNode()
}
// warn against non-primitive key
if (process.env.NODE_ENV !== 'production' &&
  isDef(data) && isDef(data.key) && !isPrimitive(data.key))
) {
  if (!__WEEEX__ || !('@binding' in data.key)) {
    warn(
      'Avoid using non-primitive value as key, ' +
      'use string/number value instead.',
      context
    )
  }
}
// support single function children as default scoped slot
if (Array.isArray(children) &&
  typeof children[0] === 'function'
) {
  data = data || {}
  data.scopedSlots = { default: children[0] }
  children.length = 0
}
if (normalizationType === ALWAYS_NORMALIZE) {
  children = normalizeChildren(children)
} else if (normalizationType === SIMPLE_NORMALIZE) {
  children = simpleNormalizeChildren(children)
}
let vnode, ns
if (typeof tag === 'string') {
  let Ctor
  ns = (context.$vnode && context.$vnode.ns) || config.getTagNamespace(tag)
  if (config.isReservedTag(tag)) {
    // platform built-in elements
    vnode = new VNode(
      config.parsePlatformTagName(tag), data, children,
      undefined, undefined, context
    )
  } else if (isDef(Ctor = resolveAsset(context.$options, 'components', tag))) {
    // component
    vnode = createComponent(Ctor, data, context, children, tag)
  } else {
    // unknown or unlisted namespaced elements
    // check at runtime because it may get assigned a namespace when its
    // parent normalizes children
    vnode = new VNode(
      tag, data, children,
      undefined, undefined, context
    )
  }
}

```

```

} else {
  // direct component options / constructor
  vnode = createComponent(tag, data, context, children)
}
if (Array.isArray(vnode)) {
  return vnode
} else if (isDef(vnode)) {
  if (isDef(ns)) applyNS(vnode, ns)
  if (isDef(data)) registerDeepBindings(data)
  return vnode
} else {
  return createEmptyVNode()
}
}
}

```

`_createElement` 方法有 5 个参数，`context` 表示 VNode 的上下文环境，它是 `Component` 类型；`tag` 表示标签，它可以是一个字符串，也可以是一个 `Component`；`data` 表示 VNode 的数据，它是一个 `VNodeData` 类型，可以在 `flow/vnode.js` 中找到它的定义，这里先不展开说；`children` 表示当前 VNode 的子节点，它是任意类型的，它接下来需要被规范为标准的 VNode 数组；`normalizationType` 表示子节点规范的类型，类型不同规范的方法也就不一样，它主要是参考 `render` 函数是编译生成的还是用户手写的。

`createElement` 函数的流程略微有点多，我们接下来主要分析 2 个重点的流程——`children` 的规范化以及 VNode 的创建。

children 的规范化

由于 Virtual DOM 实际上是一个树状结构，每一个 VNode 可能会有若干个子节点，这些子节点应该也是 VNode 的类型。`_createElement` 接收的第 4 个参数 `children` 是任意类型的，因此我们需要把它们规范成 VNode 类型。

这里根据 `normalizationType` 的不同，调用了 `normalizeChildren(children)` 和 `simpleNormalizeChildren(children)` 方法，它们的定义都在 `src/core/vdom/helpers/normalize-children.js` 中：

```

// The template compiler attempts to minimize the need for normalization by
// statically analyzing the template at compile time.
//
// For plain HTML markup, normalization can be completely skipped because the
// generated render function is guaranteed to return Array<VNode>. There are
// two cases where extra normalization is needed:

// 1. When the children contains components - because a functional component
// may return an Array instead of a single root. In this case, just a simple
// normalization is needed - if any child is an Array, we flatten the whole
// thing with Array.prototype.concat. It is guaranteed to be only 1-level deep
// because functional components already normalize their own children.
export function simpleNormalizeChildren (children: any) {

```

```

    for (let i = 0; i < children.length; i++) {
      if (Array.isArray(children[i])) {
        return Array.prototype.concat.apply([], children)
      }
    }
    return children
  }

// 2. When the children contains constructs that always generated nested Arrays,
// e.g. <template>, <slot>, v-for, or when the children is provided by user
// with hand-written render functions / JSX. In such cases a full normalization
// is needed to cater to all possible types of children values.
export function normalizeChildren (children: any): ?Array<VNode> {
  return isPrimitive(children)
  ? [createTextVNode(children)]
  : Array.isArray(children)
    ? normalizeArrayChildren(children)
    : undefined
}

```

`simpleNormalizeChildren` 方法调用场景是 `render` 函数当函数是编译生成的。理论上编译生成的 `children` 都已经是 `VNode` 类型的，但这里有一个例外，就是 `functional component` 函数式组件返回的是一个数组而不是一个根节点，所以会通过 `Array.prototype.concat` 方法把整个 `children` 数组打平，让它的深度只有一层。

`normalizeChildren` 方法的调用场景有 2 种，一个场景是 `render` 函数是用户手写的，当 `children` 只有一个节点的时候，`Vue.js` 从接口层面允许用户把 `children` 写成基础类型用来创建单个简单的文本节点，这种情况会调用 `createTextVNode` 创建一个文本节点的 `VNode`；另一个场景是当编译 `slot`、`v-for` 的时候会产生嵌套数组的情况，会调用 `normalizeArrayChildren` 方法，接下来看一下它的实现：

```

function normalizeArrayChildren (children: any, nestedIndex?: string): Array<VNode>
{
  const res = []
  let i, c, lastIndex, last
  for (i = 0; i < children.length; i++) {
    c = children[i]
    if (isUndef(c) || typeof c === 'boolean') continue
    lastIndex = res.length - 1
    last = res[lastIndex]
    // nested
    if (Array.isArray(c)) {
      if (c.length > 0) {
        c = normalizeArrayChildren(c, `${nestedIndex} || ''}_${i}`)
        // merge adjacent text nodes
        if (isTextNode(c[0]) && isTextNode(last)) {
          res[lastIndex] = createTextVNode(last.text + (c[0]: any).text)
          c.shift()
        }
      }
    }
  }
}

```

```

        res.push.apply(res, c)
    }
} else if (isPrimitive(c)) {
    if (isTextNode(last)) {
        // merge adjacent text nodes
        // this is necessary for SSR hydration because text nodes are
        // essentially merged when rendered to HTML strings
        res[lastIndex] = createTextVNode(last.text + c)
    } else if (c !== '') {
        // convert primitive to vnode
        res.push(createTextVNode(c))
    }
} else {
    if (isTextNode(c) && isTextNode(last)) {
        // merge adjacent text nodes
        res[lastIndex] = createTextVNode(last.text + c.text)
    } else {
        // default key for nested array children (likely generated by v-for)
        if (isTrue(children._isVList) &&
            isDef(c.tag) &&
            isUndef(c.key) &&
            isDef(nestedIndex)) {
            c.key = `__vlist${nestedIndex}_${i}`
        }
        res.push(c)
    }
}
return res
}

```

`normalizeArrayChildren` 接收 2 个参数，`children` 表示要规范的子节点，`nestedIndex` 表示嵌套的索引，因为单个 `child` 可能是一个数组类型。`normalizeArrayChildren` 主要的逻辑就是遍历 `children`，获得单个节点 `c`，然后对 `c` 的类型判断，如果是一个数组类型，则递归调用 `normalizeArrayChildren`；如果是基础类型，则通过 `createTextVNode` 方法转换成 VNode 类型；否则就已经是 VNode 类型了，如果 `children` 是一个列表并且列表还存在嵌套的情况，则根据 `nestedIndex` 去更新它的 key。这里需要注意一点，在遍历的过程中，对这 3 种情况都做了如下处理：如果存在两个连续的 `text` 节点，会把它们合并成一个 `text` 节点。

经过对 `children` 的规范化，`children` 变成了一个类型为 VNode 的 Array。

VNode 的创建

回到 `createElement` 函数，规范化 `children` 后，接下来会去创建一个 VNode 的实例：

```

let vnode, ns
if (typeof tag === 'string') {

```

```

let Ctor
ns = (context.$vnode && context.$vnode.ns) || config.getTagNamespace(tag)
if (config.isReservedTag(tag)) {
  // platform built-in elements
  vnode = new VNode(
    config.parsePlatformTagName(tag), data, children,
    undefined, undefined, context
  )
} else if (isDef(Ctor = resolveAsset(context.$options, 'components', tag))) {
  // component
  vnode = createComponent(Ctor, data, context, children, tag)
} else {
  // unknown or unlisted namespaced elements
  // check at runtime because it may get assigned a namespace when its
  // parent normalizes children
  vnode = new VNode(
    tag, data, children,
    undefined, undefined, context
  )
}
} else {
  // direct component options / constructor
  vnode = createComponent(tag, data, context, children)
}

```

这里先对 `tag` 做判断，如果是 `string` 类型，则接着判断如果是内置的一些节点，则直接创建一个普通 `VNode`，如果是为已注册的组件名，则通过 `createComponent` 创建一个组件类型的 `VNode`，否则创建一个未知的标签的 `VNode`。如果是 `tag` 一个 `Component` 类型，则直接调用 `createComponent` 创建一个组件类型的 `VNode` 节点。对于 `createComponent` 创建组件类型的过程，我们之后会去介绍，本质上它还是返回了一个 `VNode`。

总结

那么至此，我们大致了解了 `createElement` 创建 `VNode` 的过程，每个 `VNode` 有 `children`，`children` 每个元素也是一个 `VNode`，这样就形成了一个 `VNode Tree`，它很好的描述了我们的 `DOM Tree`。

回到 `mountComponent` 函数的过程，我们已经知道 `vm._render` 是如何创建了一个 `VNode`，接下来就是要把这个 `VNode` 渲染成一个真实的 `DOM` 并渲染出来，这个过程是通过 `vm._update` 完成的，接下来分析一下这个过程。

update

Vue 的 `_update` 是实例的一个私有方法，它被调用的时机有 2 个，一个是首次渲染，一个是数据更新的时候；由于我们这一章节只分析首次渲染部分，数据更新部分会在之后分析响应式原理的时候涉及。`_update` 方法的作用是把 VNode 渲染成真实的 DOM，它的定义在

`src/core/instance/lifecycle.js` 中：

```
Vue.prototype._update = function (vnode: VNode, hydrating?: boolean) {
  const vm: Component = this
  const prevEl = vm.$el
  const prevVnode = vm._vnode
  const prevActiveInstance = activeInstance
  activeInstance = vm
  vm._vnode = vnode
  // Vue.prototype.__patch__ is injected in entry points
  // based on the rendering backend used.
  if (!prevVnode) {
    // initial render
    vm.$el = vm.__patch__(vm.$el, vnode, hydrating, false /* removeOnly */)
  } else {
    // updates
    vm.$el = vm.__patch__(prevVnode, vnode)
  }
  activeInstance = prevActiveInstance
  // update __vue__ reference
  if (prevEl) {
    prevEl.__vue__ = null
  }
  if (vm.$el) {
    vm.$el.__vue__ = vm
  }
  // if parent is an HOC, update its $el as well
  if (vm.$vnode && vm.$parent && vm.$vnode === vm.$parent._vnode) {
    vm.$parent.$el = vm.$el
  }
  // updated hook is called by the scheduler to ensure that children are
  // updated in a parent's updated hook.
}
```

`_update` 的核心就是调用 `vm.__patch__` 方法，这个方法实际上在不同的平台，比如 web 和 weex 上的定义是不一样的，因此在 web 平台中它的定义在 `src/platforms/web/runtime/index.js` 中：

```
Vue.prototype.__patch__ = inBrowser ? patch : noop
```

可以看到，甚至在 web 平台上，是否是服务端渲染也会对这个方法产生影响。因为在服务端渲染中，没有真实的浏览器 DOM 环境，所以不需要把 VNode 最终转换成 DOM，因此是一个空函数，而在浏览器端渲染中，它指向了 `patch` 方法，它的定义在 `src/platforms/web/runtime/patch.js` 中：

```
import * as nodeOps from 'web/runtime/node-ops'
import { createPatchFunction } from 'core/vdom/patch'
import baseModules from 'core/vdom/modules/index'
import platformModules from 'web/runtime/modules/index'

// the directive module should be applied last, after all
// built-in modules have been applied.
const modules = platformModules.concat(baseModules)

export const patch: Function = createPatchFunction({ nodeOps, modules })
```

该方法的定义是调用 `createPatchFunction` 方法的返回值，这里传入了一个对象，包含 `nodeOps` 参数和 `modules` 参数。其中，`nodeOps` 封装了一系列 DOM 操作的方法，`modules` 定义了一些模块的钩子函数的实现，我们这里先不详细介绍，来看一下 `createPatchFunction` 的实现，它定义在 `src/core/vdom/patch.js` 中：

```
const hooks = ['create', 'activate', 'update', 'remove', 'destroy']

export function createPatchFunction (backend) {
  let i, j
  const cbs = {}

  const { modules, nodeOps } = backend

  for (i = 0; i < hooks.length; ++i) {
    cbs[hooks[i]] = []
    for (j = 0; j < modules.length; ++j) {
      if (isDef(modules[j][hooks[i]])) {
        cbs[hooks[i]].push(modules[j][hooks[i]])
      }
    }
  }

  // ...

  return function patch (oldVnode, vnode, hydrating, removeOnly) {
    if (isUndef(vnode)) {
      if (isDef(oldVnode)) invokeDestroyHook(oldVnode)
      return
    }

    let isInitialPatch = false
    const insertedVnodeQueue = []

    if (isUndef(oldVnode)) {
```

```

    // empty mount (likely as component), create new root element
    isInitialPatch = true
    createElm(vnode, insertedVnodeQueue)
} else {
  const isRealElement = isDef(oldVnode.nodeType)
  if (!isRealElement && sameVnode(oldVnode, vnode)) {
    // patch existing root node
    patchVnode(oldVnode, vnode, insertedVnodeQueue, removeOnly)
  } else {
    if (isRealElement) {
      // mounting to a real element
      // check if this is server-rendered content and if we can perform
      // a successful hydration.
      if (oldVnode.nodeType === 1 && oldVnode.hasAttribute(SSR_ATTR)) {
        oldVnode.removeAttribute(SSR_ATTR)
        hydrating = true
      }
      if (isTrue(hydrating)) {
        if (hydrate(oldVnode, vnode, insertedVnodeQueue)) {
          invokeInsertHook(vnode, insertedVnodeQueue, true)
          return oldVnode
        } else if (process.env.NODE_ENV !== 'production') {
          warn(
            'The client-side rendered virtual DOM tree is not matching ' +
            'server-rendered content. This is likely caused by incorrect ' +
            'HTML markup, for example nesting block-level elements inside ' +
            '<p>, or missing <tbody>. Bailing hydration and performing ' +
            'full client-side render.'
          )
        }
      }
    }
    // either not server-rendered, or hydration failed.
    // create an empty node and replace it
    oldVnode = emptyNodeAt(oldVnode)
  }

  // replacing existing element
  const oldElm = oldVnode.elm
  const parentElm = nodeOps.parentNode(oldElm)

  // create new node
  createElm(
    vnode,
    insertedVnodeQueue,
    // extremely rare edge case: do not insert if old element is in a
    // leaving transition. Only happens when combining transition +
    // keep-alive + HOCs. (#4590)
    oldElm._leaveCb ? null : parentElm,
    nodeOps.nextSibling(oldElm)
  )
}

```

```

        // update parent placeholder node element, recursively
        if (isDef(vnode.parent)) {
            let ancestor = vnode.parent
            const patchable = isPatchable(vnode)
            while (ancestor) {
                for (let i = 0; i < cbs.destroy.length; ++i) {
                    cbs.destroy[i](ancestor)
                }
                ancestor.elm = vnode.elm
                if (patchable) {
                    for (let i = 0; i < cbs.create.length; ++i) {
                        cbs.create[i](emptyNode, ancestor)
                    }
                    // #6513
                    // invoke insert hooks that may have been merged by create hooks.
                    // e.g. for directives that uses the "inserted" hook.
                    const insert = ancestor.data.hook.insert
                    if (insert.merged) {
                        // start at index 1 to avoid re-invoking component mounted hook
                        for (let i = 1; i < insert.fns.length; i++) {
                            insert.fns[i]()
                        }
                    } else {
                        registerRef(ancestor)
                    }
                    ancestor = ancestor.parent
                }
            }
        }

        // destroy old node
        if (isDef(parentElm)) {
            removeVnodes(parentElm, [oldVnode], 0, 0)
        } else if (isDef(oldVnode.tag)) {
            invokeDestroyHook(oldVnode)
        }
    }
}

invokeInsertHook(vnode, insertedVnodeQueue, isInitialPatch)
return vnode.elm
}
}

```

`createPatchFunction` 内部定义了一系列的辅助方法，最终返回了一个 `patch` 方法，这个方法就赋值给了 `vm._update` 函数里调用的 `vm.__patch__`。

在介绍 `patch` 的方法实现之前，我们可以思考一下为何 Vue.js 源码绕了这么一大圈，把相关代码分散到各个目录。因为前面介绍过，`patch` 是平台相关的，在 Web 和 Weex 环境，它们把虚拟 DOM 映射到“平台 DOM”的方法是不同的，并且对“DOM”包括的属性模块创建和更新也不尽相同。因此每个

平台都有各自的 `nodeOps` 和 `modules`，它们的代码需要托管在 `src/platforms` 这个大目录下。

而不同平台的 `patch` 的主要逻辑部分是相同的，所以这部分公共的部分托管在 `core` 这个大目录下。差异化部分只需要通过参数来区别，这里用到了一个函数柯里化的技巧，通过 `createPatchFunction` 把差异化参数提前固化，这样不用每次调用 `patch` 的时候都传递 `nodeOps` 和 `modules` 了，这种编程技巧也非常值得学习。

在这里，`nodeOps` 表示对“平台 DOM”的一些操作方法，`modules` 表示平台的一些模块，它们会在整个 `patch` 过程的不同阶段执行相应的钩子函数。这些代码的具体实现会在之后的章节介绍。

回到 `patch` 方法本身，它接收 4 个参数，`oldVnode` 表示旧的 VNode 节点，它也可以不存在或者是一个 DOM 对象；`vnode` 表示执行 `_render` 后返回的 VNode 的节点；`hydrating` 表示是否是服务端渲染；`removeOnly` 是给 `transition-group` 用的，之后会介绍。

`patch` 的逻辑看上去相对复杂，因为它有着非常多的分支逻辑，为了方便理解，我们并不会在这里介绍所有的逻辑，仅会针对我们之前的例子分析它的执行逻辑。之后我们对其它场景做源码分析的时候会再次回顾 `patch` 方法。

先来回顾我们的例子：

```
var app = new Vue({
  el: '#app',
  render: function (createElement) {
    return createElement('div', {
      attrs: {
        id: 'app'
      },
      this.message
    },
    data: {
      message: 'Hello Vue!'
    }
  }
})
```

然后我们在 `vm._update` 的方法里是这么调用 `patch` 方法的：

```
// initial render
vm.$el = vm.__patch__(vm.$el, vnode, hydrating, false /* removeOnly */)
```

结合我们的例子，我们的场景是首次渲染，所以在执行 `patch` 函数的时候，传入的 `vm.$el` 对应的是例子中 `id` 为 `app` 的 DOM 对象，这个也就是我们在 `index.html` 模板中写的 `<div id="app">`，`vm.$el` 的赋值是在之前 `mountComponent` 函数做的，`vnode` 对应的是调用 `render` 函数的返回值，`hydrating` 在非服务端渲染情况下为 `false`，`removeOnly` 为 `false`。

确定了这些入参后，我们回到 `patch` 函数的执行过程，看几个关键步骤。

```
const isRealElement = isDef(oldVnode.nodeType)
if (!isRealElement && sameVnode(oldVnode, vnode)) {
  // patch existing root node
```

```

patchVnode(oldVnode, vnode, insertedVnodeQueue, removeOnly)
} else {
  if (isRealElement) {
    // mounting to a real element
    // check if this is server-rendered content and if we can perform
    // a successful hydration.
    if (oldVnode.nodeType === 1 && oldVnode.hasAttribute(SSR_ATTR)) {
      oldVnode.removeAttribute(SSR_ATTR)
      hydrating = true
    }
    if (isTrue(hydrating)) {
      if (hydrate(oldVnode, vnode, insertedVnodeQueue)) {
        invokeInsertHook(vnode, insertedVnodeQueue, true)
        return oldVnode
      } else if (process.env.NODE_ENV !== 'production') {
        warn(
          'The client-side rendered virtual DOM tree is not matching ' +
          'server-rendered content. This is likely caused by incorrect ' +
          'HTML markup, for example nesting block-level elements inside ' +
          '<p>, or missing <tbody>. Bailing hydration and performing ' +
          'full client-side render.'
        )
      }
    }
  }
  // either not server-rendered, or hydration failed.
  // create an empty node and replace it
  oldVnode = emptyNodeAt(oldVnode)
}

// replacing existing element
const oldElm = oldVnode.elm
const parentElm = nodeOps.parentNode(oldElm)

// create new node
createElm(
  vnode,
  insertedVnodeQueue,
  // extremely rare edge case: do not insert if old element is in a
  // leaving transition. Only happens when combining transition +
  // keep-alive + HOCs. (#4590)
  oldElm._leaveCb ? null : parentElm,
  nodeOps.nextSibling(oldElm),
)
}
}

```

由于我们传入的 `oldVnode` 实际上是一个 DOM container，所以 `isRealElement` 为 `true`，接下来又通过 `emptyNodeAt` 方法把 `oldVnode` 转换成 `vNode` 对象，然后再调用 `createElm` 方法，这个方法在这里非常重要，来看一下它的实现：

```
function createElm (
```

```

vnode,
insertedVnodeQueue,
parentElm,
refElm,
nested,
ownerArray,
index
) {
if (isDef(vnode.elm) && isDef(ownerArray)) {
  // This vnode was used in a previous render!
  // now it's used as a new node, overwriting its elm would cause
  // potential patch errors down the road when it's used as an insertion
  // reference node. Instead, we clone the node on-demand before creating
  // associated DOM element for it.
  vnode = ownerArray[index] = cloneVNode(vnode)
}

vnode.isRootInsert = !nested // for transition enter check
if (createComponent(vnode, insertedVnodeQueue, parentElm, refElm)) {
  return
}

const data = vnode.data
const children = vnode.children
const tag = vnode.tag
if (isDef(tag)) {
  if (process.env.NODE_ENV !== 'production') {
    if (data && data.pre) {
      creatingElmInVPre++
    }
    if (isUnknownElement(vnode, creatingElmInVPre)) {
      warn(
        'Unknown custom element: <' + tag + '> - did you ' +
        'register the component correctly? For recursive components, ' +
        'make sure to provide the "name" option.',
        vnode.context
      )
    }
  }
}

vnode.elm = vnode.ns
? nodeOps.createElementNS(vnode.ns, tag)
: nodeOps.createElement(tag, vnode)
setScope(vnode)

/* istanbul ignore if */
if (__WEEX__) {
  // ...
} else {
  createChildren(vnode, children, insertedVnodeQueue)
  if (isDef(data)) {

```

```

        invokeCreateHooks(vnode, insertedVnodeQueue)
    }
    insert(parentElm, vnode.elm, refElm)
}

if (process.env.NODE_ENV !== 'production' && data && data.pre) {
  creatingElmInVPre--
}
} else if (isTrue(vnode.isComment)) {
  vnode.elm = nodeOps.createComment(vnode.text)
  insert(parentElm, vnode.elm, refElm)
} else {
  vnode.elm = nodeOps.createTextNode(vnode.text)
  insert(parentElm, vnode.elm, refElm)
}
}
}

```

`createElm` 的作用是通过虚拟节点创建真实的 DOM 并插入到它的父节点中。我们来看一下它的一些关键逻辑，`createComponent` 方法目的是尝试创建子组件，这个逻辑在之后组件的章节会详细介绍，在当前这个 case 下它的返回值为 false；接下来判断 `vnode` 是否包含 tag，如果包含，先简单对 tag 的合法性在非生产环境下做校验，看是否是一个合法标签；然后再去调用平台 DOM 的操作去创建一个占位符元素。

```

vnode.elm = vnode.ns
? nodeOps.createElementNS(vnode.ns, tag)
: nodeOps.createElement(tag, vnode)

```

接下来调用 `createChildren` 方法去创建子元素：

```

createChildren(vnode, children, insertedVnodeQueue)

function createChildren (vnode, children, insertedVnodeQueue) {
  if (Array.isArray(children)) {
    if (process.env.NODE_ENV !== 'production') {
      checkDuplicateKeys(children)
    }
    for (let i = 0; i < children.length; ++i) {
      createElm(children[i], insertedVnodeQueue, vnode.elm, null, true, children, i)
    }
  } else if (isPrimitive(vnode.text)) {
    nodeOps.appendChild(vnode.elm, nodeOps.createTextNode(String(vnode.text)))
  }
}

```

`createChildren` 的逻辑很简单，实际上是遍历子虚拟节点，递归调用 `createElm`，这是一种常用的深度优先的遍历算法，这里要注意的一点是在遍历过程中会把 `vnode.elm` 作为父容器的 DOM 节点占位符传入。

接着再调用 `invokeCreateHooks` 方法执行所有的 `create` 的钩子并把 `vnode` push 到 `insertedVnodeQueue` 中。

```
if (isDef(data)) {
  invokeCreateHooks(vnode, insertedVnodeQueue)
}

function invokeCreateHooks (vnode, insertedVnodeQueue) {
  for (let i = 0; i < cbs.create.length; ++i) {
    cbs.create[i](emptyNode, vnode)
  }
  i = vnode.data.hook // Reuse variable
  if (isDef(i)) {
    if (isDef(i.create)) i.create(emptyNode, vnode)
    if (isDef(i.insert)) insertedVnodeQueue.push(vnode)
  }
}
```

最后调用 `insert` 方法把 `DOM` 插入到父节点中，因为是递归调用，子元素会优先调用 `insert`，所以整个 `vnode` 树节点的插入顺序是先子后父。来看一下 `insert` 方法，它的定义在 `src/core/vdom/patch.js` 上。

```
insert(parentElm, vnode.elm, refElm)

function insert (parent, elm, ref) {
  if (isDef(parent)) {
    if (isDef(ref)) {
      if (ref.parentNode === parent) {
        nodeOps.insertBefore(parent, elm, ref)
      }
    } else {
      nodeOps.appendChild(parent, elm)
    }
  }
}
```

`insert` 逻辑很简单，调用一些 `nodeOps` 把子节点插入到父节点中，这些辅助方法定义在 `src/platforms/web/runtime/node-ops.js` 中：

```
export function insertBefore (parentNode: Node, newNode: Node, referenceNode: Node)
{
  parentNode.insertBefore(newNode, referenceNode)
}
```

```
export function appendChild (node: Node, child: Node) {
  node.appendChild(child)
}
```

其实就是在调用原生 DOM 的 API 进行 DOM 操作，看到这里，很多同学恍然大悟，原来 Vue 是这样动态创建的 DOM。

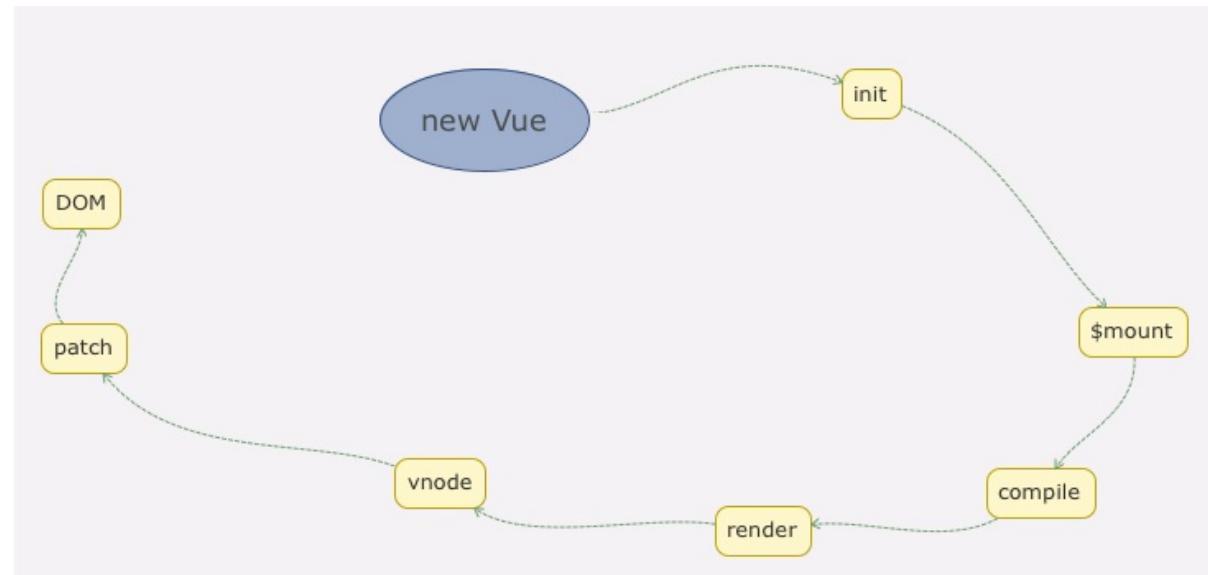
在 `createElm` 过程中，如果 `vnode` 节点如果不包含 `tag`，则它有可能是一个注释或者纯文本节点，可以直接插入到父元素中。在我们这个例子中，最内层就是一个文本 `vnode`，它的 `text` 值取的就是之前的 `this.message` 的值 `Hello Vue!`。

再回到 `patch` 方法，首次渲染我们调用了 `createElm` 方法，这里传入的 `parentElm` 是 `oldVnode.elm` 的父元素，在我们的例子是 id 为 `#app` `div` 的父元素，也就是 `Body`；实际上整个过程就是递归创建了一个完整的 DOM 树并插入到 `Body` 上。

最后，我们根据之前递归 `createElm` 生成的 `vnode` 插入顺序队列，执行相关的 `insert` 钩子函数，这部分内容我们之后会详细介绍。

总结

那么至此我们从主线上把模板和数据如何渲染成最终的 DOM 的过程分析完毕了，我们可以通过下图更直观地看到从初始化 Vue 到最终渲染的整个过程。



我们这里只是分析了最简单和最基础的场景，在实际项目中，我们是把页面拆成很多组件的，Vue 另一个核心思想就是组件化。那么下一章我们就来分析 Vue 的组件化过程。

组件化

Vue.js 另一个核心思想是组件化。所谓组件化，就是把页面拆分成多个组件 (component)，每个组件依赖的 CSS、JavaScript、模板、图片等资源放在一起开发和维护。组件是资源独立的，组件在系统内部可复用，组件和组件之间可以嵌套。

我们在用 Vue.js 开发实际项目的时候，就是像搭积木一样，编写一堆组件拼装生成页面。在 Vue.js 的官网中，也是花了大篇幅来介绍什么是组件，如何编写组件以及组件拥有的属性和特性。

那么在这一章节，我们将从源码的角度来分析 Vue 的组件内部是如何工作的，只有了解了内部的工作原理，才能让我们使用它的时候更加得心应手。

接下来我们会用 Vue-cli 初始化的代码为例，来分析一下 Vue 组件初始化的一个过程。

```
import Vue from 'vue'
import App from './App.vue'

var app = new Vue({
  el: '#app',
  // 这里的 h 是 createElement 方法
  render: h => h(App)
})
```

这段代码相信很多同学都很熟悉，它和我们上一章相同的点也是通过 `render` 函数去渲染的，不同的这次通过 `createElement` 传的参数是一个组件而不是一个原生的标签，那么接下来我们就开始分析这一过程。

createComponent

上一章我们在分析 `createElement` 的实现的时候，它最终会调用 `_createElement` 方法，其中有一段逻辑是对参数 `tag` 的判断，如果是一个普通的 html 标签，像上一章的例子那样是一个普通的 `div`，则会实例化一个普通 `VNode` 节点，否则通过 `createComponent` 方法创建一个组件 `VNode`。

```

if (typeof tag === 'string') {
  let Ctor
  ns = (context.$vnode && context.$vnode.ns) || config.getTagNamespace(tag)
  if (config.isReservedTag(tag)) {
    // platform built-in elements
    vnode = new VNode(
      config.parsePlatformTagName(tag), data, children,
      undefined, undefined, context
    )
  } else if (isDef(Ctor = resolveAsset(context.$options, 'components', tag))) {
    // component
    vnode = createComponent(Ctor, data, context, children, tag)
  } else {
    // unknown or unlisted namespaced elements
    // check at runtime because it may get assigned a namespace when its
    // parent normalizes children
    vnode = new VNode(
      tag, data, children,
      undefined, undefined, context
    )
  }
} else {
  // direct component options / constructor
  vnode = createComponent(tag, data, context, children)
}

```

在我们这一章传入的是一个 `App` 对象，它本质上是一个 `Component` 类型，那么它会走到上述代码的 `else` 逻辑，直接通过 `createComponent` 方法来创建 `vnode`。所以接下来我们来看一下 `createComponent` 方法的实现，它定义在 `src/core/vdom/create-component.js` 文件中：

```

export function createComponent (
  Ctor: Class<Component> | Function | Object | void,
  data: ?VNodeData,
  context: Component,
  children: ?Array<VNode>,
  tag?: string
): VNode | Array<VNode> | void {
  if (isUndef(Ctor)) {
    return
  }
}

```

```

const baseCtor = context.$options._base

// plain options object: turn it into a constructor
if (isObject(Ctor)) {
  Ctor = baseCtor.extend(Ctor)
}

// if at this stage it's not a constructor or an async component factory,
// reject.
if (typeof Ctor !== 'function') {
  if (process.env.NODE_ENV !== 'production') {
    warn(`Invalid Component definition: ${String(Ctor)}`, context)
  }
  return
}

// async component
let asyncFactory
if (isUndef(Ctor.cid)) {
  asyncFactory = Ctor
  Ctor = resolveAsyncComponent(asyncFactory, baseCtor, context)
  if (Ctor === undefined) {
    // return a placeholder node for async component, which is rendered
    // as a comment node but preserves all the raw information for the node.
    // the information will be used for async server-rendering and hydration.
    return createAsyncPlaceholder(
      asyncFactory,
      data,
      context,
      children,
      tag
    )
  }
}
}

data = data || {}

// resolve constructor options in case global mixins are applied after
// component constructor creation
resolveConstructorOptions(Ctor)

// transform component v-model data into props & events
if (isDef(data.model)) {
  transformModel(Ctor.options, data)
}

// extract props
const propsData = extractPropsFromVNodeData(data, Ctor, tag)

// functional component
if (isTrue(Ctor.options.functional)) {

```

```

    return createFunctionalComponent(Ctor, propsData, data, context, children)
}

// extract listeners, since these needs to be treated as
// child component listeners instead of DOM listeners
const listeners = data.on
// replace with listeners with .native modifier
// so it gets processed during parent component patch.
data.on = data.nativeOn

if (isTrue(Ctor.options.abstract)) {
  // abstract components do not keep anything
  // other than props & listeners & slot

  // work around flow
  const slot = data.slot
  data = {}
  if (slot) {
    data.slot = slot
  }
}

// install component management hooks onto the placeholder node
installComponentHooks(data)

// return a placeholder vnode
const name = Ctor.options.name || tag
const vnode = new VNode(
  `vue-component-${Ctor.cid}${name ? `-${name}` : ''}`,
  data, undefined, undefined, undefined, context,
  { Ctor, propsData, listeners, tag, children },
  asyncFactory
)

// Weex specific: invoke recycle-list optimized @render function for
// extracting cell-slot template.
// https://github.com/Hanks10100/weex-native-directive/tree/master/component
/* istanbul ignore if */
if (__WEEEX__ && isRecyclableComponent(vnode)) {
  return renderRecyclableComponentTemplate(vnode)
}

return vnode
}

```

可以看到，`createComponent` 的逻辑也会有一些复杂，但是分析源码比较推荐的是只分析核心流程，分支流程可以之后针对性的看，所以这里针对组件渲染这个 case 主要就 3 个关键步骤：

构造子类构造函数，安装组件钩子函数和实例化 `vnode`。

构造子类构造函数

```
const baseCtor = context.$options._base

// plain options object: turn it into a constructor
if (isObject(Ctor)) {
  Ctor = baseCtor.extend(Ctor)
}
```

我们在编写一个组件的时候，通常都是创建一个普通对象，还是以我们的 App.vue 为例，代码如下：

```
import HelloWorld from './components/HelloWorld'

export default {
  name: 'app',
  components: {
    HelloWorld
  }
}
```

这里 export 的是一个对象，所以 `createComponent` 里的代码逻辑会执行到 `baseCtor.extend(Ctor)`，在这里 `baseCtor` 实际上就是 `Vue`，这个的定义是在最开始初始化 `Vue` 的阶段，在 `src/core/global-api/index.js` 中的 `initGlobalAPI` 函数有这么一段逻辑：

```
// this is used to identify the "base" constructor to extend all plain-object
// components with in Weex's multi-instance scenarios.
Vue.options._base = Vue
```

细心的同学会发现，这里定义的是 `Vue.option`，而我们的 `createComponent` 取的是 `context.$options`，实际上在 `src/core/instance/init.js` 里 `Vue` 原型上的 `_init` 函数中有这么一段逻辑：

```
vm.$options = mergeOptions(
  resolveConstructorOptions(vm.constructor),
  options || {},
  vm
)
```

这样就把 `Vue` 上的一些 `option` 扩展到了 `vm.$option` 上，所以我们也就能通过 `vm.$options._base` 拿到 `Vue` 这个构造函数了。`mergeOptions` 的实现我们会在后续章节中具体分析，现在只需要理解它的功能是把 `Vue` 构造函数的 `options` 和用户传入的 `options` 做一层合并，到 `vm.$options` 上。

在了解了 `baseCtor` 指向了 `Vue` 之后，我们来看一下 `Vue.extend` 函数的定义，在 `src/core/global-api/extend.js` 中。

```
/**
```

```

    * Class inheritance
    */
Vue.extend = function (extendOptions: Object): Function {
  extendOptions = extendOptions || {}
  const Super = this
  const SuperId = Super.cid
  const cachedCtors = extendOptions._Ctor || (extendOptions._Ctor = {})
  if (cachedCtors[SuperId]) {
    return cachedCtors[SuperId]
  }

  const name = extendOptions.name || Super.options.name
  if (process.env.NODE_ENV !== 'production' && name) {
    validateComponentName(name)
  }

  const Sub = function VueComponent (options) {
    this._init(options)
  }
  Sub.prototype = Object.create(Super.prototype)
  Sub.prototype.constructor = Sub
  Sub.cid = cid++
  Sub.options = mergeOptions(
    Super.options,
    extendOptions
  )
  Sub['super'] = Super

  // For props and computed properties, we define the proxy getters on
  // the Vue instances at extension time, on the extended prototype. This
  // avoids Object.defineProperty calls for each instance created.
  if (Sub.options.props) {
    initProps(Sub)
  }
  if (Sub.options.computed) {
    initComputed(Sub)
  }

  // allow further extension/mixin/plugin usage
  Sub.extend = Super.extend
  Sub.mixin = Super.mixin
  Sub.use = Super.use

  // create asset registers, so extended classes
  // can have their private assets too.
  ASSET_TYPES.forEach(function (type) {
    Sub[type] = Super[type]
  })
  // enable recursive self-lookup
  if (name) {
    Sub.options.components[name] = Sub
  }
}

```

```

    }

    // keep a reference to the super options at extension time.
    // later at instantiation we can check if Super's options have
    // been updated.
    Sub.superOptions = Super.options
    Sub.extendOptions = extendOptions
    Sub.sealedOptions = extend({}, Sub.options)

    // cache constructor
    cachedCtors[SuperId] = Sub
    return Sub
}

```

`Vue.extend` 的作用就是构造一个 `Vue` 的子类，它使用一种非常经典的原型继承的方式把一个纯对象转换成一个继承于 `Vue` 的构造器 `Sub` 并返回，然后对 `Sub` 这个对象本身扩展了一些属性，如扩展 `options`、添加全局 API 等；并且对配置中的 `props` 和 `computed` 做了初始化工作；最后对于这个 `Sub` 构造函数做了缓存，避免多次执行 `Vue.extend` 的时候对同一个子组件重复构造。

这样当我们去实例化 `Sub` 的时候，就会执行 `this._init` 逻辑再次走到了 `Vue` 实例的初始化逻辑，实例化子组件的逻辑在之后的章节会介绍。

```

const Sub = function VueComponent (options) {
  this._init(options)
}

```

安装组件钩子函数

```

// install component management hooks onto the placeholder node
installComponentHooks(data)

```

我们之前提到 `Vue.js` 使用的 Virtual DOM 参考的是开源库 `snabbdom`，它的一个特点是在 `VNode` 的 patch 流程中对外暴露了各种时机的钩子函数，方便我们做一些额外的事情，`Vue.js` 也是充分利用这一点，在初始化一个 Component 类型的 `VNode` 的过程中实现了几个钩子函数：

```

const componentVNodeHooks = {
  init (vnode: VNodeWithData, hydrating: boolean): ?boolean {
    if (
      vnode.componentInstance &&
      !vnode.componentInstance._isDestroyed &&
      vnode.data.keepAlive
    ) {
      // kept-alive components, treat as a patch
      const mountedNode: any = vnode // work around flow
      componentVNodeHooks.prepatch(mountedNode, mountedNode)
    } else {

```

```

    const child = vnode.componentInstance = createComponentInstanceForVnode(
      vnode,
      activeInstance
    )
    child.$mount(hydrating ? vnode.elm : undefined, hydrating)
  }
),

prepatch (oldVnode: MountedComponentVNode, vnode: MountedComponentVNode) {
  const options = vnode.componentOptions
  const child = vnode.componentInstance = oldVnode.componentInstance
  updateChildComponent(
    child,
    options.propsData, // updated props
    options.listeners, // updated listeners
    vnode, // new parent vnode
    options.children // new children
  )
},

insert (vnode: MountedComponentVNode) {
  const { context, componentInstance } = vnode
  if (!componentInstance._isMounted) {
    componentInstance._isMounted = true
    callHook(componentInstance, 'mounted')
  }
  if (vnode.data.keepAlive) {
    if (context._isMounted) {
      // vue-router#1212
      // During updates, a kept-alive component's child components may
      // change, so directly walking the tree here may call activated hooks
      // on incorrect children. Instead we push them into a queue which will
      // be processed after the whole patch process ended.
      queueActivatedComponent(componentInstance)
    } else {
      activateChildComponent(componentInstance, true /* direct */)
    }
  }
},
destroy (vnode: MountedComponentVNode) {
  const { componentInstance } = vnode
  if (!componentInstance._isDestroyed) {
    if (!vnode.data.keepAlive) {
      componentInstance.$destroy()
    } else {
      deactivateChildComponent(componentInstance, true /* direct */)
    }
  }
}

```

```

const hooksToMerge = Object.keys(componentVNodeHooks)

function installComponentHooks (data: VNodeData) {
  const hooks = data.hook || (data.hook = {})
  for (let i = 0; i < hooksToMerge.length; i++) {
    const key = hooksToMerge[i]
    const existing = hooks[key]
    const toMerge = componentVNodeHooks[key]
    if (existing !== toMerge && !(existing && existing._merged)) {
      hooks[key] = existing ? mergeHook(toMerge, existing) : toMerge
    }
  }
}

function mergeHook (f1: any, f2: any): Function {
  const merged = (a, b) => {
    // flow complains about extra args which is why we use any
    f1(a, b)
    f2(a, b)
  }
  merged._merged = true
  return merged
}

```

整个 `installComponentHooks` 的过程就是把 `componentVNodeHooks` 的钩子函数合并到 `data.hook` 中，在 `VNode` 执行 `patch` 的过程中执行相关的钩子函数，具体的执行我们稍后在介绍 `patch` 过程中会详细介绍。这里要注意的是合并策略，在合并过程中，如果某个时机的钩子已经存在 `data.hook` 中，那么通过执行 `mergeHook` 函数做合并，这个逻辑很简单，就是在最终执行的时候，依次执行这两个钩子函数即可。

实例化 VNode

```

const name = Ctor.options.name || tag
const vnode = new VNode(
  `vue-component-${Ctor.cid}${name ? `-${name}` : ''}`,
  data, undefined, undefined, context,
  { Ctor, propsData, listeners, tag, children },
  asyncFactory
)
return vnode

```

最后一步非常简单，通过 `new vnode` 实例化一个 `vnode` 并返回。需要注意的是和普通元素节点的 `vnode` 不同，组件的 `vnode` 是没有 `children` 的，这点很关键，在之后的 `patch` 过程中我们会再提。

总结

这一节我们分析了 `createComponent` 的实现，了解到它在渲染一个组件的时候的 3 个关键逻辑：构造子类构造函数，安装组件钩子函数和实例化 `vnode`。`createComponent` 后返回的是组件 `vnode`，它也一样走到 `vm._update` 方法，进而执行了 `patch` 函数，我们在上一章对 `patch` 函数做了简单的分析，那么下一节我们会对它做进一步的分析。

patch

通过前一章的分析我们知道，当我们通过 `createComponent` 创建了组件 VNode，接下来会走到 `vm._update`，执行 `vm.__patch__` 去把 VNode 转换成真正的 DOM 节点。这个过程我们在前一章已经分析过了，但是针对一个普通的 VNode 节点，接下来我们来看看组件的 VNode 会有哪些不一样的地方。

patch 的过程会调用 `createElement` 创建元素节点，回顾一下 `createElement` 的实现，它的定义在 `src/core/vdom/patch.js` 中：

```
function createElement (
  vnode,
  insertedVnodeQueue,
  parentElm,
  refElm,
  nested,
  ownerArray,
  index
) {
  // ...
  if (createComponent(vnode, insertedVnodeQueue, parentElm, refElm)) {
    return
  }
  // ...
}
```

createComponent

我们删掉多余的代码，只保留关键的逻辑，这里会判断 `createComponent(vnode, insertedVnodeQueue, parentElm, refElm)` 的返回值，如果为 `true` 则直接结束，那么接下来看一下 `createComponent` 方法的实现：

```
function createComponent (vnode, insertedVnodeQueue, parentElm, refElm) {
  let i = vnode.data
  if (isDef(i)) {
    const isReactivated = isDef(vnode.componentInstance) && i.keepAlive
    if (isDef(i = i.hook) && isDef(i = i.init)) {
      i(vnode, false /* hydrating */)
    }
    // after calling the init hook, if the vnode is a child component
    // it should've created a child instance and mounted it. the child
    // component also has set the placeholder vnode's elm.
    // in that case we can just return the element and be done.
    if (isDef(vnode.componentInstance)) {
      initComponent(vnode, insertedVnodeQueue)
      insert(parentElm, vnode.elm, refElm)
    }
  }
}
```

```

        if (isTrue(isReactivated)) {
          reactivateComponent(vnode, insertedVnodeQueue, parentElm, refElm)
        }
        return true
      }
    }
}

```

`createComponent` 函数中，首先对 `vnode.data` 做了一些判断：

```

let i = vnode.data
if (isDef(i)) {
  // ...
  if (isDef(i = i.hook) && isDef(i = i.init)) {
    i(vnode, false /* hydrating */)
    // ...
  }
  // ...
}

```

如果 `vnode` 是一个组件 VNode，那么条件会满足，并且得到 `i` 就是 `init` 钩子函数，回顾上节我们在创建组件 VNode 的时候合并钩子函数中就包含 `init` 钩子函数，定义在

`src/core/vdom/create-component.js` 中：

```

init (vnode: VNodeWithData, hydrating: boolean): ?boolean {
  if (
    vnode.componentInstance &&
    !vnode.componentInstance._isDestroyed &&
    vnode.data.keepAlive
  ) {
    // kept-alive components, treat as a patch
    const mountedNode: any = vnode // work around flow
    componentVNodeHooks.prepatch(mountedNode, mountedNode)
  } else {
    const child = vnode.componentInstance = createComponentInstanceForVnode(
      vnode,
      activeInstance
    )
    child.$mount(hydrating ? vnode.elm : undefined, hydrating)
  }
},

```

`init` 钩子函数执行也很简单，我们先不考虑 `keepAlive` 的情况，它是通过 `createComponentInstanceForVnode` 创建一个 Vue 的实例，然后调用 `$mount` 方法挂载子组件，先来看一下 `createComponentInstanceForVnode` 的实现：

```

export function createComponentInstanceForVnode (
  vnode: any, // we know it's MountedComponentVNode but flow doesn't

```

```

parent: any, // activeInstance in lifecycle state
): Component {
  const options: InternalComponentOptions = {
    _isComponent: true,
    _parentVnode: vnode,
    parent
  }
  // check inline-template render functions
  const inlineTemplate = vnode.data.inlineTemplate
  if (isDef(inlineTemplate)) {
    options.render = inlineTemplate.render
    options.staticRenderFns = inlineTemplate.staticRenderFns
  }
  return new vnode.componentOptions.Ctor(options)
}

```

`createComponentInstanceForVnode` 函数构造的一个内部组件的参数，然后执行 `new vnode.componentOptions.Ctor(options)`。这里的 `vnode.componentOptions.Ctor` 对应的就是子组件的构造函数，我们上一节分析了它实际上是继承于 Vue 的一个构造器 `Sub`，相当于 `new Sub(options)`。这里有几个关键参数要注意几个点，`_isComponent` 为 `true` 表示它是一个组件，`parent` 表示当前激活的组件实例（注意，这里比较有意思的是如何拿到组件实例，后面会介绍）。

所以子组件的实例化实际上就是在这个时机执行的，并且它会执行实例的 `_init` 方法，这个过程有一些和之前不同的地方需要挑出来说明，代码在 `src/core/instance/init.js` 中：

```

Vue.prototype._init = function (options?: Object) {
  const vm: Component = this
  // merge options
  if (options && options._isComponent) {
    // optimize internal component instantiation
    // since dynamic options merging is pretty slow, and none of the
    // internal component options needs special treatment.
    initInternalComponent(vm, options)
  } else {
    vm.$options = mergeOptions(
      resolveConstructorOptions(vm.constructor),
      options || {},
      vm
    )
  }
  // ...
  if (vm.$options.el) {
    vm.$mount(vm.$options.el)
  }
}

```

这里首先是合并 `options` 的过程有变化，`_isComponent` 为 true，所以走到了 `initInternalComponent` 过程，这个函数的实现也简单看一下：

```
export function initInternalComponent (vm: Component, options: InternalComponentOptions) {
  const opts = vm.$options = Object.create(vm.constructor.options)
  // doing this because it's faster than dynamic enumeration.
  const parentVnode = options._parentVnode
  opts.parent = options.parent
  opts._parentVnode = parentVnode

  const vnodeComponentOptions = parentVnode.componentOptions
  opts.propsData = vnodeComponentOptions.propsData
  opts._parentListeners = vnodeComponentOptions.listeners
  opts._renderChildren = vnodeComponentOptions.children
  opts._componentTag = vnodeComponentOptions.tag

  if (options.render) {
    opts.render = options.render
    opts.staticRenderFns = options.staticRenderFns
  }
}
```

这个过程我们重点记住以下几个点即可：`opts.parent = options.parent`、`opts._parentVnode = parentVnode`，它们是把之前我们通过 `createComponentInstanceForVnode` 函数传入的几个参数合并到内部的选项 `$options` 里了。

再来看一下 `_init` 函数最后执行的代码：

```
if (vm.$options.el) {
  vm.$mount(vm.$options.el)
}
```

由于组件初始化的时候是不传 el 的，因此组件是自己接管了 `$mount` 的过程，这个过程的主要流程在上一章介绍过了，回到组件 `init` 的过程，`componentVNodeHooks` 的 `init` 钩子函数，在完成实例化的 `_init` 后，接着会执行 `child.$mount(hydrating ? vnode.elm : undefined, hydrating)`。这里 `hydrating` 为 true 一般是服务端渲染的情况，我们只考虑客户端渲染，所以这里 `$mount` 相当于执行 `child.$mount(undefined, false)`，它最终会调用 `mountComponent` 方法，进而执行 `vm._render()` 方法：

```
Vue.prototype._render = function (): VNode {
  const vm: Component = this
  const { render, _parentVnode } = vm.$options

  // set parent vnode. this allows render functions to have access
  // to the data on the placeholder node.
  vm.$vnode = _parentVnode
```

```
// render self
let vnode
try {
  vnode = render.call(vm._renderProxy, vm.$createElement)
} catch (e) {
  // ...
}
// set parent
vnode.parent = _parentVnode
return vnode
}
```

我们只保留关键部分的代码，这里的 `_parentVnode` 就是当前组件的父 VNode，而 `render` 函数生成的 `vnode` 当前组件的渲染 `vnode`，`vnode` 的 `parent` 指向了 `_parentVnode`，也就是 `vm.$vnode`，它们是一种父子的关系。

我们知道在执行完 `vm._render` 生成 VNode 后，接下来就要执行 `vm._update` 去渲染 VNode 了。来看一下组件渲染的过程中有哪些需要注意的，`vm._update` 的定义在 `src/core/instance/lifecycle.js` 中：

```
export let activeInstance: any = null
Vue.prototype._update = function (vnode: VNode, hydrating?: boolean) {
  const vm: Component = this
  const prevEl = vm.$el
  const prevVnode = vm._vnode
  const prevActiveInstance = activeInstance
  activeInstance = vm
  vm._vnode = vnode
  // Vue.prototype.__patch__ is injected in entry points
  // based on the rendering backend used.
  if (!prevVnode) {
    // initial render
    vm.$el = vm.__patch__(vm.$el, vnode, hydrating, false /* removeOnly */)
  } else {
    // updates
    vm.$el = vm.__patch__(prevVnode, vnode)
  }
  activeInstance = prevActiveInstance
  // update __vue__ reference
  if (prevEl) {
    prevEl.__vue__ = null
  }
  if (vm.$el) {
    vm.$el.__vue__ = vm
  }
  // if parent is an HOC, update its $el as well
  if (vm.$vnode && vm.$parent && vm.$vnode === vm.$parent._vnode) {
    vm.$parent.$el = vm.$el
  }
}
```

```
// updated hook is called by the scheduler to ensure that children are
// updated in a parent's updated hook.
}
```

`_update` 过程中有几个关键的代码，首先 `vm._vnode = vnode` 的逻辑，这个 `vnode` 是通过 `vm._render()` 返回的组件渲染 VNode，`vm._vnode` 和 `vm.$vnode` 的关系就是一种父子关系，用代码表达就是 `vm._vnode.parent === vm.$vnode`。还有一段比较有意思的代码：

```
export let activeInstance: any = null
Vue.prototype._update = function (vnode: VNode, hydrating?: boolean) {
    // ...
    const prevActiveInstance = activeInstance
    activeInstance = vm
    if (!prevVnode) {
        // initial render
        vm.$el = vm.__patch__(vm.$el, vnode, hydrating, false /* removeOnly */)
    } else {
        // updates
        vm.$el = vm.__patch__(prevVnode, vnode)
    }
    activeInstance = prevActiveInstance
}
```

这个 `activeInstance` 作用就是保持当前上下文的 Vue 实例，它是在 `lifecycle` 模块的全局变量，定义是 `export let activeInstance: any = null`，并且在之前我们调用 `createComponentInstanceForVnode` 方法的时候从 `lifecycle` 模块获取，并且作为参数传入的。因为实际上 JavaScript 是一个单线程，Vue 整个初始化是一个深度遍历的过程，在实例化子组件的过程中，它需要知道当前上下文的 Vue 实例是什么，并把它作为子组件的父 Vue 实例。之前我们提到过对子组件的实例化过程先会调用 `initInternalComponent(vm, options)` 合并 `options`，把 `parent` 存储在 `vm.$options` 中，在 `$mount` 之前会调用 `initLifecycle(vm)` 方法：

```
export function initLifecycle (vm: Component) {
    const options = vm.$options

    // locate first non-abstract parent
    let parent = options.parent
    if (parent && !options.abstract) {
        while (parent.$options.abstract && parent.$parent) {
            parent = parent.$parent
        }
        parent.$children.push(vm)
    }

    vm.$parent = parent
    // ...
}
```

可以看到 `vm.$parent` 就是用来保留当前 `vm` 的父实例，并且通过 `parent.$children.push(vm)` 来把当前的 `vm` 存储到父实例的 `$children` 中。

在 `vm._update` 的过程中，把当前的 `vm` 赋值给 `activeInstance`，同时通过 `const prevActiveInstance = activeInstance` 用 `prevActiveInstance` 保留上一次的 `activeInstance`。实际上，`prevActiveInstance` 和当前的 `vm` 是一个父子关系，当一个 `vm` 实例完成它的所有子树的 patch 或者 update 过程后，`activeInstance` 会回到它的父实例，这样就完美地保证了 `createComponentInstanceForVnode` 整个深度遍历过程中，我们在实例化子组件的时候能传入当前子组件的父 Vue 实例，并在 `_init` 的过程中，通过 `vm.$parent` 把这个父子关系保留。

那么回到 `_update`，最后就是调用 `__patch__` 渲染 VNode 了。

```
vm.$el = vm.__patch__(vm.$el, vnode, hydrating, false /* removeOnly */)

function patch (oldVnode, vnode, hydrating, removeOnly) {
  // ...
  let isInitialPatch = false
  const insertedVnodeQueue = []

  if (isUndef(oldVnode)) {
    // empty mount (likely as component), create new root element
    isInitialPatch = true
    createElm(vnode, insertedVnodeQueue)
  } else {
    // ...
  }
  // ...
}
```

这里又回到了本节开始的过程，之前分析过负责渲染成 DOM 的函数是 `createElm`，注意这里我们只传了 2 个参数，所以对应的 `parentElm` 是 `undefined`。我们再来看看它的定义：

```
function createElm (
  vnode,
  insertedVnodeQueue,
  parentElm,
  refElm,
  nested,
  ownerArray,
  index
) {
  // ...
  if (createComponent(vnode, insertedVnodeQueue, parentElm, refElm)) {
    return
  }

  const data = vnode.data
  const children = vnode.children
```

```

const tag = vnode.tag
if (isDef(tag)) {
    // ...

vnode.elm = vnode.ns
    ? nodeOps.createElementNS(vnode.ns, tag)
    : nodeOps.createElement(tag, vnode)
setScope(vnode)

/* istanbul ignore if */
if (__WEEF__) {
    // ...
} else {
    createChildren(vnode, children, insertedVnodeQueue)
    if (isDef(data)) {
        invokeCreateHooks(vnode, insertedVnodeQueue)
    }
    insert(parentElm, vnode.elm, refElm)
}

// ...
} else if (isTrue(vnode.isComment)) {
    vnode.elm = nodeOps.createComment(vnode.text)
    insert(parentElm, vnode.elm, refElm)
} else {
    vnode.elm = nodeOps.createTextNode(vnode.text)
    insert(parentElm, vnode.elm, refElm)
}
}

```

注意，这里我们传入的 `vnode` 是组件渲染的 `vnode`，也就是我们之前说的 `vm._vnode`，如果组件的根节点是个普通元素，那么 `vm._vnode` 也是普通的 `vnode`，这里 `createComponent(vnode, insertedVnodeQueue, parentElm, refElm)` 的返回值是 `false`。接下来的过程就和我们上一章一样了，先创建一个父节点占位符，然后再遍历所有子 VNode 递归调用 `createElement`，在遍历的过程中，如果遇到子 VNode 是一个组件的 VNode，则重复本节开始的过程，这样通过一个递归的方式就可以完整地构建了整个组件树。

由于我们这个时候传入的 `parentElm` 是空，所以对组件的插入，在 `createComponent` 有这么一段逻辑：

```

function createComponent (vnode, insertedVnodeQueue, parentElm, refElm) {
    let i = vnode.data
    if (isDef(i)) {
        // ...
        if (isDef(i = i.hook) && isDef(i = i.init)) {
            i(vnode, false /* hydrating */)
        }
        // ...
        if (isDef(vnode.componentInstance)) {

```

```
    initComponent(vnode, insertedVnodeQueue)
    insert(parentElm, vnode.elm, refElm)
    if (isTrue(isReactivated)) {
      reactivateComponent(vnode, insertedVnodeQueue, parentElm, refElm)
    }
    return true
  }
}
```

在完成组件的整个 `patch` 过程后，最后执行 `insert(parentElm, vnode.elm, refElm)` 完成组件的 DOM 插入，如果组件 `patch` 过程中又创建了子组件，那么 DOM 的插入顺序是先子后父。

总结

那么到此，一个组件的 VNode 是如何创建、初始化、渲染的过程也就介绍完毕了。在对组件化的实现有一个大概了解后，接下来我们来介绍一下这其中的一些细节。我们知道编写一个组件实际上是编写一个 JavaScript 对象，对象的描述就是各种配置，之前我们提到在 `_init` 的最初阶段执行的就是 `merge options` 的逻辑，那么下一节我们从源码角度来分析合并配置的过程。

合并配置

通过之前章节的源码分析我们知道，`new Vue` 的过程通常有 2 种场景，一种是外部我们的代码主动调用 `new Vue(options)` 的方式实例化一个 Vue 对象；另一种是我们上一节分析的组件过程中内部通过 `new Vue(options)` 实例化子组件。

无论哪种场景，都会执行实例的 `_init(options)` 方法，它首先会执行一个 `merge options` 的逻辑，相关的代码在 `src/core/instance/init.js` 中：

```
Vue.prototype._init = function (options?: Object) {
  // merge options
  if (options && options._isComponent) {
    // optimize internal component instantiation
    // since dynamic options merging is pretty slow, and none of the
    // internal component options needs special treatment.
    initInternalComponent(vm, options)
  } else {
    vm.$options = mergeOptions(
      resolveConstructorOptions(vm.constructor),
      options || {},
      vm
    )
  }
  // ...
}
```

可以看到不同场景对于 `options` 的合并逻辑是不一样的，并且传入的 `options` 值也有非常大的不同，接下来我会分开介绍 2 种场景的 `options` 合并过程。

为了更直观，我们可以举个简单的示例：

```
import Vue from 'vue'

let childComp = {
  template: '<div>{{msg}}</div>',
  created() {
    console.log('child created')
  },
  mounted() {
    console.log('child mounted')
  },
  data() {
    return {
      msg: 'Hello Vue'
    }
  }
}
```

```

Vue.mixin({
  created() {
    console.log('parent created')
  }
})

let app = new Vue({
  el: '#app',
  render: h => h(childComp)
})

```

外部调用场景

当执行 `new Vue` 的时候，在执行 `this._init(options)` 的时候，就会执行如下逻辑去合并 `options`：

```

vm.$options = mergeOptions(
  resolveConstructorOptions(vm.constructor),
  options || {},
  vm
)

```

这里通过调用 `mergeOptions` 方法来合并，它实际上就是把 `resolveConstructorOptions(vm.constructor)` 的返回值和 `options` 做合并，`resolveConstructorOptions` 的实现先不考虑，在我们这个场景下，它还是简单返回 `vm.constructor.options`，相当于 `Vue.options`，那么这个值又是什么呢，其实在 `initGlobalAPI(Vue)` 的时候定义了这个值，代码在 `src/core/global-api/index.js` 中：

```

export function initGlobalAPI (Vue: GlobalAPI) {
  // ...
  Vue.options = Object.create(null)
  ASSET_TYPES.forEach(type => {
    Vue.options[type + 's'] = Object.create(null)
  })

  // this is used to identify the "base" constructor to extend all plain-object
  // components with in Weex's multi-instance scenarios.
  Vue.options._base = Vue

  extend(Vue.options.components, builtInComponents)
  // ...
}

```

首先通过 `Vue.options = Object.create(null)` 创建一个空对象，然后遍历 `ASSET_TYPES`，`ASSET_TYPES` 的定义在 `src/shared/constants.js` 中：

```
export const ASSET_TYPES = [
  'component',
  'directive',
  'filter'
]
```

所以上面遍历 `ASSET_TYPES` 后的代码相当于：

```
Vue.options.components = {}
Vue.options.directives = {}
Vue.options.filters = {}
```

接着执行了 `Vue.options._base = Vue`，它的作用在我们上节实例化子组件的时候介绍了。

最后通过 `extend(Vue.options.components, builtInComponents)` 把一些内置组件扩展到 `Vue.options.components` 上，`Vue` 的内置组件目前有 `<keep-alive>`、`<transition>` 和 `<transition-group>` 组件，这也就是为什么我们在其它组件中使用 `<keep-alive>` 组件不需要注册的原因，这块儿后续我们介绍 `<keep-alive>` 组件的时候会详细讲。

那么回到 `mergeOptions` 这个函数，它的定义在 `src/core/util/options.js` 中：

```
/**
 * Merge two option objects into a new one.
 * Core utility used in both instantiation and inheritance.
 */
export function mergeOptions (
  parent: Object,
  child: Object,
  vm?: Component
): Object {
  if (process.env.NODE_ENV !== 'production') {
    checkComponents(child)
  }

  if (typeof child === 'function') {
    child = child.options
  }

  normalizeProps(child, vm)
  normalizeInject(child, vm)
  normalizeDirectives(child)
  const extendsFrom = child.extends
  if (extendsFrom) {
    parent = mergeOptions(parent, extendsFrom, vm)
  }
  if (child.mixins) {
    for (let i = 0, l = child.mixins.length; i < l; i++) {
      parent = mergeOptions(parent, child.mixins[i], vm)
    }
  }
}
```

```

    }
    const options = {}
    let key
    for (key in parent) {
        mergeField(key)
    }
    for (key in child) {
        if (!hasOwn(parent, key)) {
            mergeField(key)
        }
    }
    function mergeField (key) {
        const strat = strats[key] || defaultStrat
        options[key] = strat(parent[key], child[key], vm, key)
    }
    return options
}

```

`mergeOptions` 主要功能就是把 `parent` 和 `child` 这两个对象根据一些合并策略，合并成一个新对象并返回。比较核心的几步，先递归把 `extends` 和 `mixixns` 合并到 `parent` 上，然后遍历 `parent`，调用 `mergeField`，然后再遍历 `child`，如果 `key` 不在 `parent` 的自身属性上，则调用 `mergeField`。

这里有意思的是 `mergeField` 函数，它对不同的 `key` 有着不同的合并策略。举例来说，对于生命周期函数，它的合并策略是这样的：

```

function mergeHook (
    parentVal: ?Array<Function>,
    childVal: ?Function | ?Array<Function>
): ?Array<Function> {
    return childVal
        ? parentVal
            ? parentVal.concat(childVal)
            : Array.isArray(childVal)
                ? childVal
                : [childVal]
        : parentVal
}

LIFECYCLE_HOOKS.forEach(hook => {
    strats[hook] = mergeHook
})

```

这其中的 `LIFECYCLE_HOOKS` 的定义在 `src/shared/constants.js` 中：

```

export const LIFECYCLE_HOOKS = [
    'beforeCreate',
    'created',
    'beforeMount',

```

```
'mounted',
'beforeUpdate',
'updated',
'beforeDestroy',
'destroyed',
'activated',
'deactivated',
'errorCaptured'
]
```

这里定义了 Vue.js 所有的钩子函数名称，所以对于钩子函数，他们的合并策略都是 `mergeHook` 函数。这个函数的实现也非常有意思，用了一个多层 3 元运算符，逻辑就是如果不存在 `childVal`，就返回 `parentVal`；否则再判断是否存在 `parentVal`，如果存在就把 `childVal` 添加到 `parentVal` 后返回新数组；否则返回 `childVal` 的数组。所以回到 `mergeOptions` 函数，一旦 `parent` 和 `child` 都定义了相同的钩子函数，那么它们会把 2 个钩子函数合并成一个数组。

关于其它属性的合并策略的定义都可以在 `src/core/util/options.js` 文件中看到，这里不一一介绍了，感兴趣的同學可以自己看。

通过执行 `mergeField` 函数，把合并后的结果保存到 `options` 对象中，最终返回它。

因此，在我们当前这个 case 下，执行完如下合并后：

```
vm.$options = mergeOptions(
  resolveConstructorOptions(vm.constructor),
  options || {},
  vm
)
```

`vm.$options` 的值差不多是如下这样：

```
vm.$options = {
  components: { },
  created: [
    function created() {
      console.log('parent created')
    }
  ],
  directives: { },
  filters: { },
  _base: function Vue(options) {
    // ...
  },
  el: "#app",
  render: function (h) {
    // ...
  }
}
```

组件场景

由于组件的构造函数是通过 `Vue.extend` 继承自 `Vue` 的，先回顾一下这个过程，代码定义在 `src/core/global-api/extend.js` 中。

```
/**  
 * Class inheritance  
 */  
Vue.extend = function (extendOptions: Object): Function {  
    // ...  
    Sub.options = mergeOptions(  
        Super.options,  
        extendOptions  
    )  
  
    // ...  
    // keep a reference to the super options at extension time.  
    // later at instantiation we can check if Super's options have  
    // been updated.  
    Sub.superOptions = Super.options  
    Sub.extendOptions = extendOptions  
    Sub.sealedOptions = extend({}, Sub.options)  
  
    // ...  
    return Sub  
}
```

我们只保留关键逻辑，这里的 `extendOptions` 对应的就是前面定义的组件对象，它会和 `Vue.options` 合并到 `Sub.options` 中。

接下来我们再回忆一下子组件的初始化过程，代码定义在 `src/core/vdom/create-component.js` 中：

```
export function createComponentInstanceForVnode (  
    vnode: any, // we know it's MountedComponentVNode but flow doesn't  
    parent: any, // activeInstance in lifecycle state  
): Component {  
    const options: InternalComponentOptions = {  
        _isComponent: true,  
        _parentVnode: vnode,  
        parent  
    }  
    // ...  
    return new vnode.componentOptions.Ctor(options)  
}
```

这里的 `vnode.componentOptions.Ctor` 就是指向 `Vue.extend` 的返回值 `Sub`，所以执行 `new vnode.componentOptions.Ctor(options)` 接着执行 `this._init(options)`，因为 `options._isComponent` 为 `true`，那么合并 `options` 的过程走到了 `initInternalComponent(vm, options)` 逻辑。先来看一下它的代码实现，在 `src/core/instance/init.js` 中：

```
export function initInternalComponent (vm: Component, options: InternalComponentOptions) {
  const opts = vm.$options = Object.create(vm.constructor.options)
  // doing this because it's faster than dynamic enumeration.
  const parentVnode = options._parentVnode
  opts.parent = options.parent
  opts._parentVnode = parentVnode

  const vnodeComponentOptions = parentVnode.componentOptions
  opts.propsData = vnodeComponentOptions.propsData
  opts._parentListeners = vnodeComponentOptions.listeners
  opts._renderChildren = vnodeComponentOptions.children
  opts._componentTag = vnodeComponentOptions.tag

  if (options.render) {
    opts.render = options.render
    opts.staticRenderFns = options.staticRenderFns
  }
}
```

`initInternalComponent` 方法首先执行 `const opts = vm.$options = Object.create(vm.constructor.options)`，这里的 `vm.construction` 就是子组件的构造函数 `Sub`，相当于 `vm.$options = Sub.options`。

接着又把实例化子组件传入的子组件父 VNode 实例 `parentVnode`、子组件的父 Vue 实例 `parent` 保存到 `vm.$options` 中，另外还保留了 `parentVnode` 配置中的如 `propsData` 等其它的属性。

这么看来，`initInternalComponent` 只是做了简单一层对象赋值，并不涉及到递归、合并策略等复杂逻辑。

因此，在我们当前这个 case 下，执行完如下合并后：

```
initInternalComponent(vm, options)
```

`vm.$options` 的值差不多是如下这样：

```
vm.$options = {
  parent: Vue /*父Vue实例*/,
  propsData: undefined,
  _componentTag: undefined,
  _parentVnode: VNode /*父VNode实例*/,
  _renderChildren:undefined,
  __proto__: {
```

```

components: { },
directives: { },
filters: { },
_base: function Vue(options) {
    //...
},
_ctor: {},
created: [
    function created() {
        console.log('parent created')
    }, function created() {
        console.log('child created')
    }
],
mounted: [
    function mounted() {
        console.log('child mounted')
    }
],
data() {
    return {
        msg: 'Hello Vue'
    }
},
template: '<div>{{msg}}</div>'
}
}

```

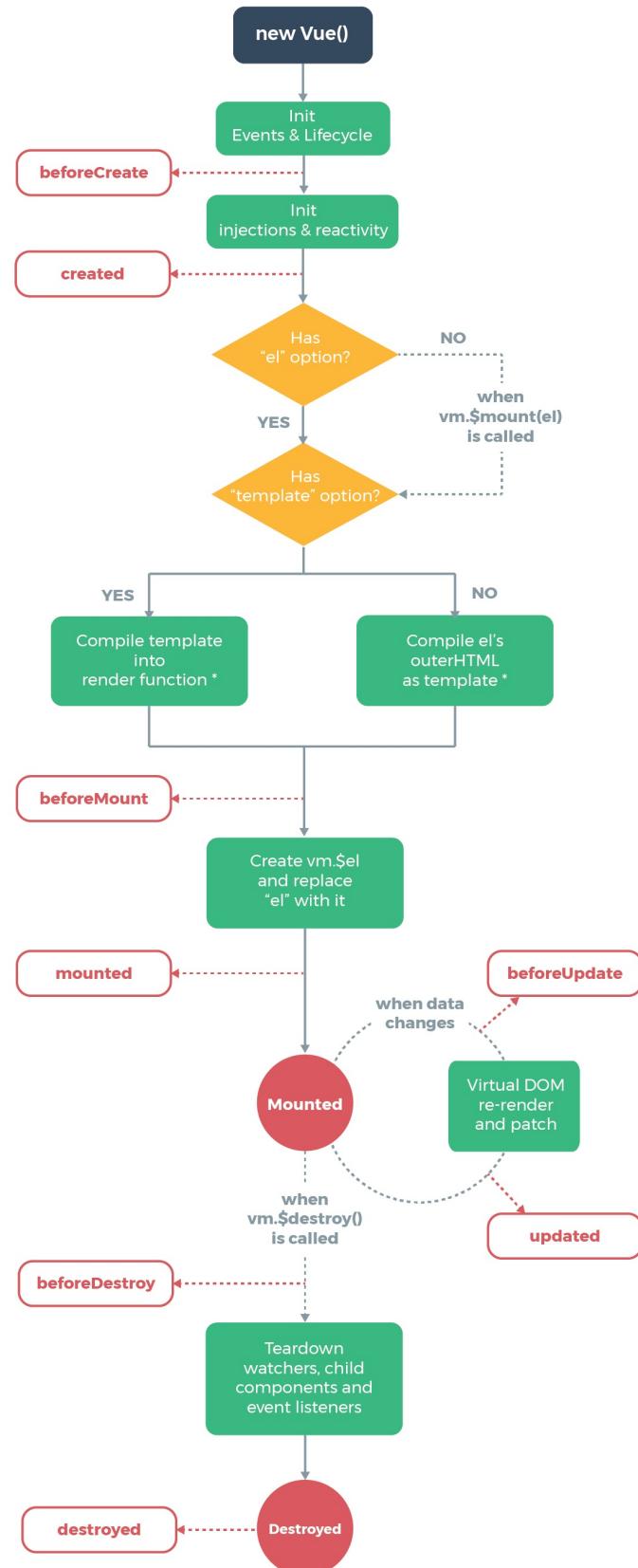
总结

那么至此，Vue 初始化阶段对于 `options` 的合并过程就介绍完了，我们需要知道对于 `options` 的合并有 2 种方式，子组件初始化过程通过 `initInternalComponent` 方式要比外部初始化 Vue 通过 `mergeOptions` 的过程要快，合并完的结果保留在 `vm.$options` 中。

纵观一些库、框架的设计几乎都是类似的，自身定义了一些默认配置，同时又可以在初始化阶段传入一些定义配置，然后去 `merge` 默认配置，来达到定制化不同需求的目的。只不过在 Vue 的场景下，会对 `merge` 的过程做一些精细化控制，虽然我们在开发自己的 JSSDK 的时候并没有 Vue 这么复杂，但这个设计思想是值得我们借鉴的。

生命周期

每个 Vue 实例在被创建之前都要经过一系列的初始化过程。例如需要设置数据监听、编译模板、挂载实例到 DOM、在数据变化时更新 DOM 等。同时在这个过程中也会运行一些叫做生命周期钩子的函数，给予用户机会在一些特定的场景下添加他们自己的代码。



在我们实际项目开发过程中，会非常频繁地和 Vue 组件的生命周期打交道，接下来我们就从源码的角度来看一下这些生命周期的钩子函数是如何被执行的。

源码中最终执行生命周期的函数都是调用 `callHook` 方法，它的定义在 `src/core/instance/lifecycle` 中：

```
export function callHook (vm: Component, hook: string) {
  // #7573 disable dep collection when invoking lifecycle hooks
  pushTarget()
  const handlers = vm.$options[hook]
  if (handlers) {
    for (let i = 0, j = handlers.length; i < j; i++) {
      try {
        handlers[i].call(vm)
      } catch (e) {
        handleError(e, vm, `${hook} hook`)
      }
    }
  }
  if (vm._hasHookEvent) {
    vm.$emit('hook:' + hook)
  }
  popTarget()
}
```

`callHook` 函数的逻辑很简单，根据传入的字符串 `hook`，去拿到 `vm.$options[hook]` 对应的回调函数数组，然后遍历执行，执行的时候把 `vm` 作为函数执行的上下文。

在上一节中，我们详细地介绍了 Vue.js 合并 `options` 的过程，各个阶段的生命周期的函数也被合并到 `vm.$options` 里，并且是一个数组。因此 `callhook` 函数的功能就是调用某个生命周期钩子注册的所有回调函数。

了解了生命周期的执行方式后，接下来我们会具体介绍每一个生命周期函数它的调用时机。

beforeCreate & created

`beforeCreate` 和 `created` 函数都是在实例化 `Vue` 的阶段，在 `_init` 方法中执行的，它的定义在 `src/core/instance/init.js` 中：

```
Vue.prototype._init = function (options?: Object) {
  // ...
  initLifecycle(vm)
  initEvents(vm)
  initRender(vm)
  callHook(vm, 'beforeCreate')
  initInjections(vm) // resolve injections before data/props
  initState(vm)
  initProvide(vm) // resolve provide after data/props
```

```

    callHook(vm, 'created')
    // ...
}

```

可以看到 `beforeCreate` 和 `created` 的钩子调用是在 `initState` 的前后，`initState` 的作用是初始化 `props`、`data`、`methods`、`watch`、`computed` 等属性，之后我们会详细分析。那么显然 `beforeCreate` 的钩子函数中就不能获取到 `props`、`data` 中定义的值，也不能调用 `methods` 中定义的函数。

在这两个钩子函数执行的时候，并没有渲染 DOM，所以我们也不能够访问 DOM，一般来说，如果组件在加载的时候需要和后端有交互，放在这两个钩子函数执行都可以，如果是需要访问 `props`、`data` 等数据的话，就需要使用 `created` 钩子函数。之后我们会介绍 vue-router 和 vuex 的时候会发现它们都混合了 `beforeCreate` 钩子函数。

beforeMount & mounted

顾名思义，`beforeMount` 钩子函数发生在 `mount`，也就是 DOM 挂载之前，它的调用时机是在 `mountComponent` 函数中，定义在 `src/core/instance/lifecycle.js` 中：

```

export function mountComponent (
  vm: Component,
  el: ?Element,
  hydrating?: boolean
): Component {
  vm.$el = el
  // ...
  callHook(vm, 'beforeMount')

  let updateComponent
  /* istanbul ignore if */
  if (process.env.NODE_ENV !== 'production' && config.performance && mark) {
    updateComponent = () => {
      const name = vm._name
      const id = vm._uid
      const startTag = `vue-perf-start:${id}`
      const endTag = `vue-perf-end:${id}`

      mark(startTag)
      const vnode = vm._render()
      mark(endTag)
      measure(`vue ${name} render`, startTag, endTag)

      mark(startTag)
      vm._update(vnode, hydrating)
      mark(endTag)
      measure(`vue ${name} patch`, startTag, endTag)
    }
  } else {

```

```

updateComponent = () => {
  vm._update(vm._render(), hydrating)
}

// we set this to vm._watcher inside the watcher's constructor
// since the watcher's initial patch may call $forceUpdate (e.g. inside child
// component's mounted hook), which relies on vm._watcher being already defined
new Watcher(vm, updateComponent, noop, {
  before () {
    if (vm._isMounted) {
      callHook(vm, 'beforeUpdate')
    }
  }
}, true /* isRenderWatcher */)

hydrating = false

// manually mounted instance, call mounted on self
// mounted is called for render-created child components in its inserted hook
if (vm.$vnode == null) {
  vm._isMounted = true
  callHook(vm, 'mounted')
}
return vm
}

```

在执行 `vm._render()` 函数渲染 VNode 之前，执行了 `beforeMount` 钩子函数，在执行完 `vm._update()` 把 VNode patch 到真实 DOM 后，执行 `mounted` 钩子。注意，这里对 `mounted` 钩子函数执行有一个判断逻辑，`vm.$vnode` 如果为 `null`，则表明这不是一次组件的初始化过程，而是我们通过外部 `new Vue` 初始化过程。那么对于组件，它的 `mounted` 时机在哪儿呢？

之前我们提到过，组件的 VNode patch 到 DOM 后，会执行 `invokeInsertHook` 函数，把 `insertedVnodeQueue` 里保存的钩子函数依次执行一遍，它的定义在 `src/core/vdom/patch.js` 中：

```

function invokeInsertHook (vnode, queue, initial) {
  // delay insert hooks for component root nodes, invoke them after the
  // element is really inserted
  if (isTrue(initial) && isDef(vnode.parent)) {
    vnode.parent.data.pendingInsert = queue
  } else {
    for (let i = 0; i < queue.length; ++i) {
      queue[i].data.hook.insert(queue[i])
    }
  }
}

```

该函数会执行 `insert` 这个钩子函数，对于组件而言，`insert` 钩子函数的定义在 `src/core/vdom/create-component.js` 中的 `componentVNodeHooks` 中：

```

const componentVNodeHooks = {
  // ...
  insert (vnode: MountedComponentVNode) {
    const { context, componentInstance } = vnode
    if (!componentInstance._isMounted) {
      componentInstance._isMounted = true
      callHook(componentInstance, 'mounted')
    }
    // ...
  },
}

```

我们可以看到，每个子组件都是在这个钩子函数中执行 `mounted` 钩子函数，并且我们之前分析过，`insertedVnodeQueue` 的添加顺序是先子后父，所以对于同步渲染的子组件而言，`mounted` 钩子函数的执行顺序也是先子后父。

beforeUpdate & updated

顾名思义，`beforeUpdate` 和 `updated` 的钩子函数执行时机都应该是数据更新的时候，到目前为止，我们还没有分析 Vue 的数据双向绑定、更新相关，下一章我会详细介绍这个过程。

`beforeUpdate` 的执行时机是在渲染 Watcher 的 `before` 函数中，我们刚才提到过：

```

export function mountComponent (
  vm: Component,
  el: ?Element,
  hydrating?: boolean
): Component {
  // ...

  // we set this to vm._watcher inside the watcher's constructor
  // since the watcher's initial patch may call $forceUpdate (e.g. inside child
  // component's mounted hook), which relies on vm._watcher being already defined
  new Watcher(vm, updateComponent, noop, {
    before () {
      if (vm._isMounted) {
        callHook(vm, 'beforeUpdate')
      }
    }
  }, true /* isRenderWatcher */)
  // ...
}

```

注意这里有个判断，也就是在组件已经 `mounted` 之后，才会去调用这个钩子函数。

`update` 的执行时机是在 `flushSchedulerQueue` 函数调用的时候，它的定义在 `src/core/observer/scheduler.js` 中：

```

function flushSchedulerQueue () {
  // ...
  // 获取到 updatedQueue
  callUpdatedHooks(updatedQueue)
}

function callUpdatedHooks (queue) {
  let i = queue.length
  while (i--) {
    const watcher = queue[i]
    const vm = watcher.vm
    if (vm._watcher === watcher && vm._isMounted) {
      callHook(vm, 'updated')
    }
  }
}

```

`flushSchedulerQueue` 函数我们之后会详细介绍，可以先大概了解一下，`updatedQueue` 是更新了的 `watcher` 数组，那么在 `callUpdatedHooks` 函数中，它对这些数组做遍历，只有满足当前 `watcher` 为 `vm._watcher` 以及组件已经 `mounted` 这两个条件，才会执行 `updated` 钩子函数。

我们之前提过，在组件 `mount` 的过程中，会实例化一个渲染的 `watcher` 去监听 `vm` 上的数据变化重新渲染，这断逻辑发生在 `mountComponent` 函数执行的时候：

```

export function mountComponent (
  vm: Component,
  el: ?Element,
  hydrating?: boolean
): Component {
  // ...
  // 这里是简写
  let updateComponent = () => {
    vm._update(vm._render(), hydrating)
  }
  new Watcher(vm, updateComponent, noop, {
    before () {
      if (vm._isMounted) {
        callHook(vm, 'beforeUpdate')
      }
    },
    true /* isRenderWatcher */
  })
}

```

那么在实例化 `watcher` 的过程中，在它的构造函数里会判断 `isRenderWatcher`，接着把当前 `watcher` 的实例赋值给 `vm._watcher`，定义在 `src/core/observer/watcher.js` 中：

```

export default class Watcher {

```

```
// ...
constructor (
  vm: Component,
  expOrFn: string | Function,
  cb: Function,
  options?: ?Object,
  isRenderWatcher?: boolean
) {
  this.vm = vm
  if (isRenderWatcher) {
    vm._watcher = this
  }
  vm._watchers.push(this)
  // ...
}
}
```

同时，还把当前 `watcher` 实例 push 到 `vm._watchers` 中，`vm._watcher` 是专门用来监听 `vm` 上数据变化然后重新渲染的，所以它是一个渲染相关的 `watcher`，因此在 `callUpdatedHooks` 函数中，只有 `vm._watcher` 的回调执行完毕后，才会执行 `updated` 钩子函数。

beforeDestroy & destroyed

顾名思义，`beforeDestroy` 和 `destroyed` 钩子函数的执行时机在组件销毁的阶段，组件的销毁过程之后会详细介绍，最终会调用 `$destroy` 方法，它的定义在 `src/core/instance/lifecycle.js` 中：

```
Vue.prototype.$destroy = function () {
  const vm: Component = this
  if (vm._isBeingDestroyed) {
    return
  }
  callHook(vm, 'beforeDestroy')
  vm._isBeingDestroyed = true
  // remove self from parent
  const parent = vm.$parent
  if (parent && !parent._isBeingDestroyed && !vm.$options.abstract) {
    remove(parent.$children, vm)
  }
  // teardown watchers
  if (vm._watcher) {
    vm._watcher.teardown()
  }
  let i = vm._watchers.length
  while (i--) {
    vm._watchers[i].teardown()
  }
  // remove reference from data ob
```

```

// frozen object may not have observer.
if (vm._data.__ob__) {
  vm._data.__ob__.vmCount--
}
// call the last hook...
vm._isDestroyed = true
// invoke destroy hooks on current rendered tree
vm.__patch__(vm._vnode, null)
// fire destroyed hook
callHook(vm, 'destroyed')
// turn off all instance listeners.
vm.$off()
// remove __vue__ reference
if (vm.$el) {
  vm.$el.__vue__ = null
}
// release circular reference (#6759)
if (vm.$vnode) {
  vm.$vnode.parent = null
}
}

```

`beforeDestroy` 钩子函数的执行时机是在 `$destroy` 函数执行最开始的地方，接着执行了一系列的销毁动作，包括从 `parent` 的 `$children` 中删掉自身，删除 `watcher`，当前渲染的 VNode 执行销毁钩子函数等，执行完毕后再调用 `destroy` 钩子函数。

在 `$destroy` 的执行过程中，它又会执行 `vm.__patch__(vm._vnode, null)` 触发它子组件的销毁钩子函数，这样一层层的递归调用，所以 `destroy` 钩子函数执行顺序是先子后父，和 `mounted` 过程一样。

activated & deactivated

`activated` 和 `deactivated` 钩子函数是专门为 `keep-alive` 组件定制的钩子，我们会在介绍 `keep-alive` 组件的时候详细介绍，这里先留个悬念。

总结

这一节主要介绍了 Vue 生命周期中各个钩子函数的执行时机以及顺序，通过分析，我们知道在 `created` 钩子函数中可以访问到数据，在 `mounted` 钩子函数中可以访问到 DOM，在 `destroy` 钩子函数中可以做一些定时器销毁工作，了解它们有利于我们在合适的生命周期去做不同的事情。

组件注册

在 Vue.js 中，除了它内置的组件如 `keep-alive`、`component`、`transition`、`transition-group` 等，其它用户自定义组件在使用前必须注册。很多同学在开发过程中可能会遇到如下报错信息：

```
'Unknown custom element: <xxxx> - did you register the component correctly?
For recursive components, make sure to provide the "name" option.'
```

一般报这个错的原因都是我们使用了未注册的组件。Vue.js 提供了 2 种组件的注册方式，全局注册和局部注册。接下来我们从源码分析的角度来分析这两种注册方式。

全局注册

要注册一个全局组件，可以使用 `Vue.component(tagName, options)`。例如：

```
Vue.component('my-component', {
  // 选项
})
```

那么，`Vue.component` 函数是在什么时候定义的呢，它的定义过程发生在最开始初始化 Vue 的全局函数的时候，代码在 `src/core/global-api/assets.js` 中：

```
import { ASSET_TYPES } from 'shared/constants'
import { isPlainObject, validateComponentName } from '../util/index'

export function initAssetRegisters (Vue: GlobalAPI) {
  /**
   * Create asset registration methods.
   */
  ASSET_TYPES.forEach(type => {
    Vue[type] = function (
      id: string,
      definition: Function | Object
    ): Function | Object | void {
      if (!definition) {
        return this.options[type + 's'][id]
      } else {
        /* istanbul ignore if */
        if (process.env.NODE_ENV !== 'production' && type === 'component') {
          validateComponentName(id)
        }
        if (type === 'component' && isPlainObject(definition)) {
          definition.name = definition.name || id
          definition = this.options._base.extend(definition)
        }
      }
    }
  })
}
```

```

        }
        if (type === 'directive' && typeof definition === 'function') {
            definition = { bind: definition, update: definition }
        }
        this.options[type + 's'][id] = definition
        return definition
    }
}
)
}

```

函数首先遍历 `ASSET_TYPES`，得到 `type` 后挂载到 `Vue` 上。`ASSET_TYPES` 的定义在 `src/shared/constants.js` 中：

```

export const ASSET_TYPES = [
    'component',
    'directive',
    'filter'
]

```

所以实际上 `Vue` 是初始化了 3 个全局函数，并且如果 `type` 是 `component` 且 `definition` 是一个对象的话，通过 `this.options._base.extend`，相当于 `Vue.extend` 把这个对象转换成一个继承于 `Vue` 的构造函数，最后通过 `this.options[type + 's'][id] = definition` 把它挂载到 `Vue.options.components` 上。

由于我们每个组件的创建都是通过 `Vue.extend` 继承而来，我们之前分析过在继承的过程中有这么一段逻辑：

```

Sub.options = mergeOptions(
    Super.options,
    extendOptions
)

```

也就是说它会把 `Vue.options` 合并到 `Sub.options`，也就是组件的 `options` 上，然后在组件的实例化阶段，会执行 `merge options` 逻辑，把 `Sub.options.components` 合并到 `vm.$options.components` 上。

然后在创建 `vnode` 的过程中，会执行 `_createElement` 方法，我们再来看看这部分的逻辑，它的定义在 `src/core/vdom/create-element.js` 中：

```

export function _createElement (
    context: Component,
    tag?: string | Class<Component> | Function | Object,
    data?: VNodeData,
    children?: any,
    normalizationType?: number
): VNode | Array<VNode> {
    // ...
}

```

```

let vnode, ns
if (typeof tag === 'string') {
  let Ctor
  ns = (context.$vnode && context.$vnode.ns) || config.getTagNamespace(tag)
  if (config.isReservedTag(tag)) {
    // platform built-in elements
    vnode = new VNode(
      config.parsePlatformTagName(tag), data, children,
      undefined, undefined, context
    )
  } else if (isDef(Ctor = resolveAsset(context.$options, 'components', tag))) {
    // component
    vnode = createComponent(Ctor, data, context, children, tag)
  } else {
    // unknown or unlisted namespaced elements
    // check at runtime because it may get assigned a namespace when its
    // parent normalizes children
    vnode = new VNode(
      tag, data, children,
      undefined, undefined, context
    )
  }
} else {
  // direct component options / constructor
  vnode = createComponent(tag, data, context, children)
}
// ...
}

```

这里有一个判断逻辑 `isDef(Ctor = resolveAsset(context.$options, 'components', tag))`，先来看一下 `resolveAsset` 的定义，在 `src/core/utils/options.js` 中：

```

/**
 * Resolve an asset.
 * This function is used because child instances need access
 * to assets defined in its ancestor chain.
 */
export function resolveAsset (
  options: Object,
  type: string,
  id: string,
  warnMissing?: boolean
): any {
  /* istanbul ignore if */
  if (typeof id !== 'string') {
    return
  }
  const assets = options[type]
  // check local registration variations first
  if (hasOwn(assets, id)) return assets[id]
}

```

```

const camelizedId = camelize(id)
if (hasOwn(assets, camelizedId)) return assets[camelizedId]
const PascalCaseId = capitalize(camelizedId)
if (hasOwn(assets, PascalCaseId)) return assets[PascalCaseId]
// fallback to prototype chain
const res = assets[id] || assets[camelizedId] || assets[PascalCaseId]
if (process.env.NODE_ENV !== 'production' && warnMissing && !res) {
  warn(
    'Failed to resolve ' + type.slice(0, -1) + ':' + id,
    options
  )
}
return res
}

```

这段逻辑很简单，先通过 `const assets = options[type]` 拿到 `assets`，然后再尝试拿 `assets[id]`，这里有个顺序，先直接使用 `id` 拿，如果不存在，则把 `id` 变成驼峰的形式再拿，如果仍然不存在则在驼峰的基础上把首字母再变成大写的形式再拿，如果仍然拿不到则报错。这样说明了我们在使用 `Vue.component(id, definition)` 全局注册组件的时候，`id` 可以是连字符、驼峰或首字母大写的形式。

那么回到我们的调用 `resolveAsset(context.$options, 'components', tag)`，即拿 `vm.$options.components[tag]`，这样我们就可以在 `resolveAsset` 的时候拿到这个组件的构造函数，并作为 `createComponent` 的钩子的参数。

局部注册

Vue.js 也同样支持局部注册，我们可以在一个组件内部使用 `components` 选项做组件的局部注册，例如：

```

import HelloWorld from './components/HelloWorld'

export default {
  components: {
    HelloWorld
  }
}

```

其实理解了全局注册的过程，局部注册是非常简单的。在组件的 Vue 的实例化阶段有一个合并 `option` 的逻辑，之前我们也分析过，所以就把 `components` 合并到 `vm.$options.components` 上，这样我们就可以在 `resolveAsset` 的时候拿到这个组件的构造函数，并作为 `createComponent` 的钩子的参数。

注意，局部注册和全局注册不同的是，只有该类型的组件才可以访问局部注册的子组件，而全局注册是扩展到 `Vue.options.components` 下，所以在所有组件创建的过程中，都会从全局的 `Vue.options.components` 扩展到当前组件的 `vm.$options.components` 下，这就是全局注册的组件能被任意使用的原因。

总结

通过这一小节的分析，我们对组件的注册过程有了认识，并理解了全局注册和局部注册的差异。其实在平时的工作中，当我们使用到组件库的时候，往往更通用基础组件都是全局注册的，而编写的特例场景的业务组件都是局部注册的。了解了它们的原理，对我们在工作中到底使用全局注册组件还是局部注册组件是有非常好的指导意义的。

异步组件

在我们平时的开发工作中，为了减少首屏代码体积，往往会被一些非首屏的组件设计成异步组件，按需加载。Vue 也原生支持了异步组件的能力，如下：

```
Vue.component('async-example', function (resolve, reject) {
  // 这个特殊的 require 语法告诉 webpack
  // 自动将编译后的代码分割成不同的块,
  // 这些块将通过 Ajax 请求自动下载。
  require(['./my-async-component'], resolve)
})
```

示例中可以看到，Vue 注册的组件不再是一个对象，而是一个工厂函数，函数有两个参数 `resolve` 和 `reject`，函数内部用 `setTimeout` 模拟了异步，实际使用可能是通过动态请求异步组件的 JS 地址，最终通过执行 `resolve` 方法，它的参数就是我们的异步组件对象。

在了解了异步组件如何注册后，我们从源码的角度来分析一下它的实现。

上一节我们分析了组件的注册逻辑，由于组件的定义并不是一个普通对象，所以不会执行 `Vue.extend` 的逻辑把它变成一个组件的构造函数，但是它仍然可以执行到 `createComponent` 函数，我们再来对这个函数做回顾，它的定义在 `src/core/vdom/create-component/js` 中：

```
export function createComponent (
  Ctor: Class<Component> | Function | Object | void,
  data: ?VNodeData,
  context: Component,
  children: ?Array<VNode>,
  tag?: string
): VNode | Array<VNode> | void {
  if (isUndef(Ctor)) {
    return
  }

  const baseCtor = context.$options._base

  // plain options object: turn it into a constructor
  if (isObject(Ctor)) {
    Ctor = baseCtor.extend(Ctor)
  }

  // ...

  // async component
  let asyncFactory
  if (isUndef(Ctor.cid)) {
    asyncFactory = Ctor
    Ctor = resolveAsyncComponent(asyncFactory, baseCtor, context)
  }
}
```

```

    if (Ctor === undefined) {
      // return a placeholder node for async component, which is rendered
      // as a comment node but preserves all the raw information for the node.
      // the information will be used for async server-rendering and hydration.
      return createAsyncPlaceholder(
        asyncFactory,
        data,
        context,
        children,
        tag
      )
    }
  }
}

```

我们省略了不必要的逻辑，只保留关键逻辑，由于我们这个时候传入的 `Ctor` 是一个函数，那么它也并不会执行 `Vue.extend` 逻辑，因此它的 `cid` 是 `undefined`，进入了异步组件创建的逻辑。这里首先执行了 `Ctor = resolveAsyncComponent(asyncFactory, baseCtor, context)` 方法，它的定义在 `src/core/vdom/helpers/resolve-async-component.js` 中：

```

export function resolveAsyncComponent (
  factory: Function,
  baseCtor: Class<Component>,
  context: Component
): Class<Component> | void {
  if (isTrue(factory.error) && isDef(factory.errorComp)) {
    return factory.errorComp
  }

  if (isDef(factory.resolved)) {
    return factory.resolved
  }

  if (isTrue(factory.loading) && isDef(factory.loadingComp)) {
    return factory.loadingComp
  }

  if (isDef(factory.contexts)) {
    // already pending
    factory.contexts.push(context)
  } else {
    const contexts = factory.contexts = [context]
    let sync = true

    const forceRender = () => {
      for (let i = 0, l = contexts.length; i < l; i++) {
        contexts[i].$forceUpdate()
      }
    }
  }
}

```

```

const resolve = once((res: object | Class<Component>) => {
  // cache resolved
  factory.resolved = ensureCtor(res, baseCtor)
  // invoke callbacks only if this is not a synchronous resolve
  // (async resolves are shimmed as synchronous during SSR)
  if (!sync) {
    forceRender()
  }
})

const reject = once(reason => {
  process.env.NODE_ENV !== 'production' && warn(
    `Failed to resolve async component: ${String(factory)} ` +
    (reason ? `\nReason: ${reason}` : '')
  )
  if (isDef(factory.errorComp)) {
    factory.error = true
    forceRender()
  }
})

const res = factory(resolve, reject)

if (isObject(res)) {
  if (typeof res.then === 'function') {
    // () => Promise
    if (isUndef(factory.resolved)) {
      res.then(resolve, reject)
    }
  } else if (isDef(res.component) && typeof res.component.then === 'function')
  {
    res.component.then(resolve, reject)

    if (isDef(res.error)) {
      factory.errorComp = ensureCtor(res.error, baseCtor)
    }

    if (isDef(res.loading)) {
      factory.loadingComp = ensureCtor(res.loading, baseCtor)
      if (res.delay === 0) {
        factory.loading = true
      } else {
        setTimeout(() => {
          if (isUndef(factory.resolved) && isUndef(factory.error)) {
            factory.loading = true
            forceRender()
          }
        }, res.delay || 200)
      }
    }
  }
}

```

```

    if (isDef(res.timeout)) {
      setTimeout(() => {
        if (isUndef(factory.resolved)) {
          reject(
            process.env.NODE_ENV !== 'production'
              ? `timeout (${res.timeout}ms)`
              : null
          )
        }
      }, res.timeout)
    }
  }

sync = false
// return in case resolved synchronously
return factory.loading
  ? factory.loadingComp
  : factory.resolved
}
}

```

`resolveAsyncComponent` 函数的逻辑略复杂，因为它实际上处理了 3 种异步组件的创建方式，除了刚才示例的组件注册方式，还支持 2 种，一种是支持 `Promise` 创建组件的方式，如下：

```

Vue.component(
  'async-webpack-example',
  // 该 `import` 函数返回一个 `Promise` 对象。
  () => import('./my-async-component')
)

```

另一种是高级异步组件，如下：

```

const AsyncComp = () => ({
  // 需要加载的组件。应当是一个 Promise
  component: import('./MyComp.vue'),
  // 加载中应当渲染的组件
  loading: LoadingComp,
  // 出错时渲染的组件
  error: ErrorComp,
  // 渲染加载中组件前的等待时间。默认：200ms。
  delay: 200,
  // 最长等待时间。超出此时间则渲染错误组件。默认：Infinity
  timeout: 3000
})
Vue.component('async-example', AsyncComp)

```

那么解下来，我们就根据这 3 种异步组件的情况，来分别去分析 `resolveAsyncComponent` 的逻辑。

普通函数异步组件

针对普通函数的情况，前面几个 if 判断可以忽略，它们是为高级组件所用，对于 `factory.contexts` 的判断，是考虑到多个地方同时初始化一个异步组件，那么它的实际加载应该只有一次。接着进入实际加载逻辑，定义了 `forceRender`、`resolve` 和 `reject` 函数，注意 `resolve` 和 `reject` 函数用 `once` 函数做了一层包装，它的定义在 `src/shared/util.js` 中：

```
/** 
 * Ensure a function is called only once.
 */
export function once (fn: Function): Function {
  let called = false
  return function () {
    if (!called) {
      called = true
      fn.apply(this, arguments)
    }
  }
}
```

`once` 逻辑非常简单，传入一个函数，并返回一个新函数，它非常巧妙地利用闭包和一个标志位保证了它包装的函数只会执行一次，也就是确保 `resolve` 和 `reject` 函数只执行一次。

接下来执行 `const res = factory(resolve, reject)` 逻辑，这块儿就是执行我们组件的工厂函数，同时把 `resolve` 和 `reject` 函数作为参数传入，组件的工厂函数通常会先发送请求去加载我们的异步组件的 JS 文件，拿到组件定义的对象 `res` 后，执行 `resolve(res)` 逻辑，它会先执行

`factory.resolved = ensureCtor(res, baseCtor) :`

```
function ensureCtor (comp: any, base) {
  if (
    comp.__esModule ||
    (hasSymbol && comp[Symbol.toStringTag] === 'Module')
  ) {
    comp = comp.default
  }
  return isObject(comp)
    ? base.extend(comp)
    : comp
}
```

这个函数目的是为了保证能找到异步组件 JS 定义的组件对象，并且如果它是一个普通对象，则调用 `vue.extend` 把它转换成一个组件的构造函数。

`resolve` 逻辑最后判断了 `sync`，显然我们这个场景下 `sync` 为 `false`，那么就会执行 `forceRender` 函数，它会遍历 `factory.contexts`，拿到每一个调用异步组件的实例 `vm`，执行 `vm.$forceUpdate()` 方法，它的定义在 `src/core/instance/lifecycle.js` 中：

```
Vue.prototype.$forceUpdate = function () {
  const vm: Component = this
  if (vm._watcher) {
    vm._watcher.update()
  }
}
```

`$forceUpdate` 的逻辑非常简单，就是调用渲染 `watcher` 的 `update` 方法，让渲染 `watcher` 对应的回调函数执行，也就是触发了组件的重新渲染。之所以这么做是因为 Vue 通常是数据驱动视图重新渲染，但是在整个异步组件加载过程中是没有数据发生变化的，所以通过执行 `$forceUpdate` 可以强制组件重新渲染一次。

Promise 异步组件

```
Vue.component(
  'async-webpack-example',
  // 该 `import` 函数返回一个 `Promise` 对象。
  () => import('./my-async-component')
)
```

webpack 2+ 支持了异步加载的语法糖：`() => import('./my-async-component')`，当执行完 `res = factory(resolve, reject)`，返回的值就是 `import('./my-async-component')` 的返回值，它是一个 `Promise` 对象。接着进入 `if` 条件，又判断了 `typeof res.then === 'function'`，条件满足，执行：

```
if (isUndef(factory.resolved)) {
  res.then(resolve, reject)
}
```

当组件异步加载成功后，执行 `resolve`，加载失败则执行 `reject`，这样就非常巧妙地实现了配合 webpack 2+ 的异步加载组件的方式（`Promise`）加载异步组件。

高级异步组件

由于异步加载组件需要动态加载 JS，有一定网络延时，而且有加载失败的情况，所以通常我们在开发异步组件相关逻辑的时候需要设计 `loading` 组件和 `error` 组件，并在适当的时机渲染它们。Vue.js 2.3+ 支持了一种高级异步组件的方式，它通过一个简单的对象配置，帮你搞定 `loading` 组件和 `error` 组件的渲染时机，你完全不用关心细节，非常方便。接下来我们就从源码的角度来分析高级异步组件是怎么实现的。

```

const AsyncComp = () => ({
  // 需要加载的组件。应当是一个 Promise
  component: import('./MyComp.vue'),
  // 加载中应当渲染的组件
  loading: LoadingComp,
  // 出错时渲染的组件
  error: ErrorComp,
  // 渲染加载中组件前的等待时间。默认：200ms。
  delay: 200,
  // 最长等待时间。超出此时间则渲染错误组件。默认：Infinity
  timeout: 3000
})
Vue.component('async-example', AsyncComp)

```

高级异步组件的初始化逻辑和普通异步组件一样，也是执行 `resolveAsyncComponent`，当执行完 `res = factory(resolve, reject)`，返回值就是定义的组件对象，显然满足 `else if (isDef(res.component) && typeof res.component.then === 'function')` 的逻辑，接着执行 `res.component.then(resolve, reject)`，当异步组件加载成功后，执行 `resolve`，失败执行 `reject`。

因为异步组件加载是一个异步过程，它接着又同步执行了如下逻辑：

```

if (isDef(res.error)) {
  factory.errorComp = ensureCtor(res.error, baseCtor)
}

if (isDef(res.loading)) {
  factory.loadingComp = ensureCtor(res.loading, baseCtor)
  if (res.delay === 0) {
    factory.loading = true
  } else {
    setTimeout(() => {
      if (isUndef(factory.resolved) && isUndef(factory.error)) {
        factory.loading = true
        forceRender()
      }
    }, res.delay || 200)
  }
}

if (isDef(res.timeout)) {
  setTimeout(() => {
    if (isUndef(factory.resolved)) {
      reject(
        process.env.NODE_ENV !== 'production'
        ? `timeout (${res.timeout}ms)`
        : null
      )
    }
  })
}

```

```
    }, res.timeout)
}
```

先判断 `res.error` 是否定义了 error 组件，如果有的话则赋值给 `factory.errorComp`。接着判断 `res.loading` 是否定义了 loading 组件，如果有的话则赋值给 `factory.loadingComp`，如果设置了 `res.delay` 且为 0，则设置 `factory.loading = true`，否则延时 `delay` 的时间执行：

```
if (isUndef(factory.resolved) && isUndef(factory.error)) {
  factory.loading = true
  forceRender()
}
```

最后判断 `res.timeout`，如果配置了该项，则在 `res.timeout` 时间后，如果组件没有成功加载，执行 `reject`。

在 `resolveAsyncComponent` 的最后有一段逻辑：

```
sync = false
return factory.loading
? factory.loadingComp
: factory.resolved
```

如果 `delay` 配置为 0，则这次直接渲染 loading 组件，否则则延时 `delay` 执行 `forceRender`，那么又会再一次执行到 `resolveAsyncComponent`。

那么这时候我们有几种情况，按逻辑的执行顺序，对不同的情况做判断。

异步组件加载失败

当异步组件加载失败，会执行 `reject` 函数：

```
const reject = once(reason => {
  process.env.NODE_ENV !== 'production' && warn(
    `Failed to resolve async component: ${String(factory)}` +
    (reason ? `\nReason: ${reason}` : '')
  )
  if (isDef(factory.errorComp)) {
    factory.error = true
    forceRender()
  }
})
```

这个时候会把 `factory.error` 设置为 `true`，同时执行 `forceRender()` 再次执行到 `resolveAsyncComponent`：

```
if (isTrue(factory.error) && isDef(factory.errorComp)) {
  return factory.errorComp
```

```
}
```

那么这个时候就返回 `factory.errorComp`，直接渲染 error 组件。

异步组件加载成功

当异步组件加载成功，会执行 `resolve` 函数：

```
const resolve = once((res: Object | Class<Component>) => {
  factory.resolved = ensureCtor(res, baseCtor)
  if (!sync) {
    forceRender()
  }
})
```

首先把加载结果缓存到 `factory.resolved` 中，这个时候因为 `sync` 已经为 `false`，则执行 `forceRender()` 再次执行到 `resolveAsyncComponent`：

```
if (isDef(factory.resolved)) {
  return factory.resolved
}
```

那么这个时候直接返回 `factory.resolved`，渲染成功加载的组件。

异步组件加载中

如果异步组件加载中并未返回，这时候会走到这个逻辑：

```
if (isTrue(factory.loading) && isDef(factory.loadingComp)) {
  return factory.loadingComp
}
```

那么则会返回 `factory.loadingComp`，渲染 loading 组件。

异步组件加载超时

如果超时，则走到了 `reject` 逻辑，之后逻辑和加载失败一样，渲染 error 组件。

异步组件 patch

回到 `createComponent` 的逻辑：

```
Ctor = resolveAsyncComponent(asyncFactory, baseCtor, context)
if (Ctor === undefined) {
  return createAsyncPlaceholder(
```

```

    asyncFactory,
    data,
    context,
    children,
    tag
)
}

```

如果是第一次执行 `resolveAsyncComponent`，除非使用高级异步组件 `o delay` 去创建了一个 `loading` 组件，否则返回是 `undefined`，接着通过 `createAsyncPlaceholder` 创建一个注释节点作为占位符。它的定义在 `src/core/vdom/helpers/resolve-async-components.js` 中：

```

export function createAsyncPlaceholder (
  factory: Function,
  data: ?VNodeData,
  context: Component,
  children: ?Array<VNode>,
  tag: ?string
): VNode {
  const node = createEmptyVNode()
  node.asyncFactory = factory
  node.asyncMeta = { data, context, children, tag }
  return node
}

```

实际上就是创建了一个占位的注释 `VNode`，同时把 `asyncFactory` 和 `asyncMeta` 赋值给当前 `vnode`。

当执行 `forceRender` 的时候，会触发组件的重新渲染，那么会再一次执行 `resolveAsyncComponent`，这时候就会根据不同的情况，可能返回 `loading`、`error` 或成功加载的异步组件，返回值不为 `undefined`，因此就走正常的组件 `render`、`patch` 过程，与组件第一次渲染流程不一样，这个时候是存在新旧 `vnode` 的，下一章我会分析组件更新的 `patch` 过程。

总结

通过以上代码分析，我们对 Vue 的异步组件的实现有了深入的了解，知道了 3 种异步组件的实现方式，并且看到高级异步组件的实现是非常巧妙的，它实现了 `loading`、`resolve`、`reject`、`timeout` 4 种状态。异步组件实现的本质是 2 次渲染，除了 `0 delay` 的高级异步组件第一次直接渲染成 `loading` 组件外，其它都是第一次渲染生成一个注释节点，当异步获取组件成功后，再通过 `forceRender` 强制重新渲染，这样就能正确渲染出我们异步加载的组件了。

深入响应式原理

前面 2 章介绍的都是 Vue 怎么实现数据渲染和组件化的，主要讲的是初始化的过程，把原始的数据最终映射到 DOM 中，但并没有涉及到数据变化到 DOM 变化的部分。而 Vue 的数据驱动除了数据渲染 DOM 之外，还有一个很重要的体现就是数据的变更会触发 DOM 的变化。

其实前端开发最重要的 2 个工作，一个是把数据渲染到页面，另一个是处理用户交互。Vue 把数据渲染到页面的能力我们已经通过源码分析出其中的原理了，但是由于一些用户交互或者是其它方面导致数据发生变化重新对页面渲染的原理我们还未分析。

考虑如下示例：

```
<div id="app" @click="changeMsg">
  {{ message }}
</div>
```

```
var app = new Vue({
  el: '#app',
  data: {
    message: 'Hello Vue!'
  },
  methods: {
    changeMsg() {
      this.message = 'Hello World!'
    }
  }
})
```

当我们去修改 `this.message` 的时候，模板对应的插值也会渲染成新的数据，那么这一切是怎么做到的呢？

在分析前，我们先直观的想一下，如果不用 Vue 的话，我们会通过最简单的方法实现这个需求：监听点击事件，修改数据，手动操作 DOM 重新渲染。这个过程和使用 Vue 的最大区别就是多了一步“手动操作 DOM 重新渲染”。这一步看上去并不多，但它背后又潜在的几个要处理的问题：

1. 我需要修改哪块的 DOM？
2. 我的修改效率和性能是不是最优的？
3. 我需要对数据每一次的修改都去操作 DOM 吗？
4. 我需要 case by case 去写修改 DOM 的逻辑吗？

如果我们使用了 Vue，那么上面几个问题 Vue 内部就帮你做了，那么 Vue 是如何在我们对数据修改后自动做这些事情呢，接下来我们将进入一些 Vue 响应式系统的底层的细节。

响应式对象

可能很多小伙伴之前都了解过 Vue.js 实现响应式的核心是利用了 ES5 的 `Object.defineProperty`，这也是为什么 Vue.js 不能兼容 IE8 及以下浏览器的原因，我们先来对它有个直观的认识。

Object.defineProperty

`Object.defineProperty` 方法会直接在一个对象上定义一个新属性，或者修改一个对象的现有属性，并返回这个对象，先来看一下它的语法：

```
Object.defineProperty(obj, prop, descriptor)
```

`obj` 是要在其上定义属性的对象；`prop` 是要定义或修改的属性的名称；`descriptor` 是将被定义或修改的属性描述符。

比较核心的是 `descriptor`，它有很多可选键值，具体的可以去参阅它的[文档](#)。这里我们最关心的是 `get` 和 `set`，`get` 是一个给属性提供的 getter 方法，当我们访问了该属性的时候会触发 `getter` 方法；`set` 是一个给属性提供的 setter 方法，当我们对该属性做修改的时候会触发 `setter` 方法。

一旦对象拥有了 `getter` 和 `setter`，我们可以简单地把这个对象称为响应式对象。那么 Vue.js 把哪些对象变成了响应式对象了呢，接下来我们从源码层面分析。

initState

在 Vue 的初始化阶段，`_init` 方法执行的时候，会执行 `initState(vm)` 方法，它的定义在 `src/core/instance/state.js` 中。

```
export function initState (vm: Component) {
  vm._watchers = []
  const opts = vm.$options
  if (opts.props) initProps(vm, opts.props)
  if (opts.methods) initMethods(vm, opts.methods)
  if (opts.data) {
    initData(vm)
  } else {
    observe(vm._data = {}, true /* asRootData */)
  }
  if (opts.computed) initComputed(vm, opts.computed)
  if (opts.watch && opts.watch !== nativeWatch) {
    initWatch(vm, opts.watch)
  }
}
```

`initState` 方法主要是对 `props`、`methods`、`data`、`computed` 和 `watcher` 等属性做了初始化操作。这里我们重点分析 `props` 和 `data`，对于其它属性的初始化我们之后再详细分析。

- `initProps`

```
function initProps (vm: Component, propsOptions: Object) {
  const propsData = vm.$options.propsData || {}
  const props = vm._props = {}
  // cache prop keys so that future props updates can iterate using Array
  // instead of dynamic object key enumeration.
  const keys = vm.$options._propKeys = []
  const isRoot = !vm.$parent
  // root instance props should be converted
  if (!isRoot) {
    toggleObserving(false)
  }
  for (const key in propsOptions) {
    keys.push(key)
    const value = validateProp(key, propsOptions, propsData, vm)
    /* istanbul ignore else */
    if (process.env.NODE_ENV !== 'production') {
      const hyphenatedKey = hyphenate(key)
      if (isReservedAttribute(hyphenatedKey) ||
          config.isReservedAttr(hyphenatedKey)) {
        warn(
          `${hyphenatedKey}` + " is a reserved attribute and cannot be used as component prop.",
          vm
        )
      }
      defineReactive(props, key, value, () => {
        if (vm.$parent && !isUpdatingChildComponent) {
          warn(
            `Avoid mutating a prop directly since the value will be overwritten whenever the parent component re-renders. ` +
            `Instead, use a data or computed property based on the prop's value. Prop being mutated: "${key}"`,
            vm
          )
        }
      })
    } else {
      defineReactive(props, key, value)
    }
    // static props are already proxied on the component's prototype
    // during Vue.extend(). We only need to proxy props defined at
    // instantiation here.
    if (!(key in vm)) {
      proxy(vm, '_props', key)
    }
  }
}
```

```

    }
    toggleObserving(true)
}

```

`props` 的初始化主要过程，就是遍历定义的 `props` 配置。遍历的过程主要做两件事情：一个是调用 `defineReactive` 方法把每个 `prop` 对应的值变成响应式，可以通过 `vm._props.xxx` 访问到定义 `props` 中对应的属性。对于 `defineReactive` 方法，我们稍后会介绍；另一个是通过 `proxy` 把 `vm._props.xxx` 的访问代理到 `vm.xxxx` 上，我们稍后也会介绍。

- `initData`

```

function initData (vm: Component) {
  let data = vm.$options.data
  data = vm._data = typeof data === 'function'
    ? getData(data, vm)
    : data || {}
  if (!isPlainObject(data)) {
    data = {}
    process.env.NODE_ENV !== 'production' && warn(
      'data functions should return an object:\n' +
      'https://vuejs.org/v2/guide/components.html#data-Must-Be-a-Function',
      vm
    )
  }
  // proxy data on instance
  const keys = Object.keys(data)
  const props = vm.$options.props
  const methods = vm.$options.methods
  let i = keys.length
  while (i--) {
    const key = keys[i]
    if (process.env.NODE_ENV !== 'production') {
      if (methods && hasOwn(methods, key)) {
        warn(
          `Method "${key}" has already been defined as a data property.`,
          vm
        )
      }
    }
    if (props && hasOwn(props, key)) {
      process.env.NODE_ENV !== 'production' && warn(
        `The data property "${key}" is already declared as a prop. ` +
        `Use prop default value instead.`,
        vm
      )
    } else if (!isReserved(key)) {
      proxy(vm, `_data`, key)
    }
  }
  // observe data
}

```

```
    observe(data, true /* asRootData */)
}
```

`data` 的初始化主要过程也是做两件事，一个是对定义 `data` 函数返回对象的遍历，通过 `proxy` 把每一个值 `vm._data.xxx` 都代理到 `vm.xxx` 上；另一个是调用 `observe` 方法观测整个 `data` 的变化，把 `data` 也变成响应式，可以通过 `vm._data.xxx` 访问到定义 `data` 返回函数中对应的属性，`observe` 我们稍后会介绍。

可以看到，无论是 `props` 或是 `data` 的初始化都是把它们变成响应式对象，这个过程我们接触到几个函数，接下来我们来详细分析它们。

proxy

首先介绍一下代理，代理的作用是把 `props` 和 `data` 上的属性代理到 `vm` 实例上，这也就是为什么比如我们定义了如下 `props`，却可以通过 `vm` 实例访问到它。

```
let comp = {
  props: {
    msg: 'hello'
  },
  methods: {
    say() {
      console.log(this.msg)
    }
  }
}
```

我们可以在 `say` 函数中通过 `this.msg` 访问到我们定义在 `props` 中的 `msg`，这个过程发生在 `proxy` 阶段：

```
const sharedPropertyDefinition = {
  enumerable: true,
  configurable: true,
  get: noop,
  set: noop
}

export function proxy (target: Object, sourceKey: string, key: string) {
  sharedPropertyDefinition.get = function proxyGetter () {
    return this[sourceKey][key]
  }
  sharedPropertyDefinition.set = function proxySetter (val) {
    this[sourceKey][key] = val
  }
  Object.defineProperty(target, key, sharedPropertyDefinition)
}
```

`proxy` 方法的实现很简单，通过 `Object.defineProperty` 把 `target[sourceKey][key]` 的读写变成了对 `target[key]` 的读写。所以对于 `props` 而言，对 `vm._props.xxx` 的读写变成了 `vm.xxx` 的读写，而对于 `vm._props.xxx` 我们可以访问到定义在 `props` 中的属性，所以我们就可以通过 `vm.xxx` 访问到定义在 `props` 中的 `xxx` 属性了。同理，对于 `data` 而言，对 `vm._data.xxxxx` 的读写变成了对 `vm.xxxxx` 的读写，而对于 `vm._data.xxxxx` 我们可以访问到定义在 `data` 函数返回对象中的属性，所以我们就可以通过 `vm.xxxxx` 访问到定义在 `data` 函数返回对象中的 `xxxx` 属性了。

observe

`observe` 的功能就是用来监测数据的变化，它的定义在 `src/core/observer/index.js` 中：

```
/**
 * Attempt to create an observer instance for a value,
 * returns the new observer if successfully observed,
 * or the existing observer if the value already has one.
 */
export function observe (value: any, asRootData: ?boolean): Observer | void {
  if (!isObject(value) || value instanceof VNode) {
    return
  }
  let ob: Observer | void
  if (hasOwn(value, '__ob__') && value.__ob__ instanceof Observer) {
    ob = value.__ob__
  } else if (
    shouldObserve &&
    !isServerRendering() &&
    (Array.isArray(value) || isPlainObject(value)) &&
    Object.isExtensible(value) &&
    !value._isVue
  ) {
    ob = new Observer(value)
  }
  if (asRootData && ob) {
    ob.vmCount++
  }
  return ob
}
```

`observe` 方法的作用就是给非 `VNode` 的对象类型数据添加一个 `Observer`，如果已经添加过则直接返回，否则在满足一定条件下去实例化一个 `Observer` 对象实例。接下来我们来看一下 `Observer` 的作用。

Observer

`Observer` 是一个类，它的作用是给对象的属性添加 `getter` 和 `setter`，用于依赖收集和派发更新：

```

/**
 * Observer class that is attached to each observed
 * object. Once attached, the observer converts the target
 * object's property keys into getter/setters that
 * collect dependencies and dispatch updates.
 */
export class Observer {
  value: any;
  dep: Dep;
  vmCount: number; // number of vms that has this object as root $data

  constructor (value: any) {
    this.value = value
    this.dep = new Dep()
    this.vmCount = 0
    def(value, '__ob__', this)
    if (Array.isArray(value)) {
      const augment = hasProto
        ? protoAugment
        : copyAugment
      augment(value, arrayMethods, arrayKeys)
      this.observeArray(value)
    } else {
      this.walk(value)
    }
  }

  /**
   * Walk through each property and convert them into
   * getter/setters. This method should only be called when
   * value type is Object.
   */
  walk (obj: Object) {
    const keys = Object.keys(obj)
    for (let i = 0; i < keys.length; i++) {
      defineReactive(obj, keys[i])
    }
  }

  /**
   * Observe a list of Array items.
   */
  observeArray (items: Array<any>) {
    for (let i = 0, l = items.length; i < l; i++) {
      observe(items[i])
    }
  }
}

```

`Observer` 的构造函数逻辑很简单，首先实例化 `Dep` 对象，这块稍后会介绍，接着通过执行 `def` 函数把自身实例添加到数据对象 `value` 的 `__ob__` 属性上，`def` 的定义在 `src/core/util/lang.js` 中：

```
/** 
 * Define a property.
 */
export function def (obj: Object, key: string, val: any, enumerable?: boolean) {
  Object.defineProperty(obj, key, {
    value: val,
    enumerable: !!enumerable,
    writable: true,
    configurable: true
  })
}
```

`def` 函数是一个非常简单的 `Object.defineProperty` 的封装，这就是为什么我在开发中输出 `data` 上对象类型的数据，会发现该对象多了一个 `__ob__` 的属性。

回到 `Observer` 的构造函数，接下来会对 `value` 做判断，对于数组会调用 `observeArray` 方法，否则对纯对象调用 `walk` 方法。可以看到 `observeArray` 是遍历数组再次调用 `observe` 方法，而 `walk` 方法是遍历对象的 `key` 调用 `defineReactive` 方法，那么我们来看一下这个方法是做什么的。

defineReactive

`defineReactive` 的功能就是定义一个响应式对象，给对象动态添加 `getter` 和 `setter`，它的定义在 `src/core/observer/index.js` 中：

```
/** 
 * Define a reactive property on an Object.
 */
export function defineReactive (
  obj: Object,
  key: string,
  val: any,
  customSetter?: ?Function,
  shallow?: boolean
) {
  const dep = new Dep()

  const property = Object.getOwnPropertyDescriptor(obj, key)
  if (property && property.configurable === false) {
    return
  }

  // cater for pre-defined getter/setters
  const getter = property && property.get
```

```

const setter = property && property.set
if ((!getter || setter) && arguments.length === 2) {
  val = obj[key]
}

let childOb = !shallow && observe(val)
Object.defineProperty(obj, key, {
  enumerable: true,
  configurable: true,
  get: function reactiveGetter () {
    const value = getter ? getter.call(obj) : val
    if (Dep.target) {
      dep.depend()
      if (childOb) {
        childOb.dep.depend()
        if (Array.isArray(value)) {
          dependArray(value)
        }
      }
    }
    return value
  },
  set: function reactiveSetter (newVal) {
    const value = getter ? getter.call(obj) : val
    /* eslint-disable no-self-compare */
    if (newVal === value || (newVal !== newVal && value !== value)) {
      return
    }
    /* eslint-enable no-self-compare */
    if (process.env.NODE_ENV !== 'production' && customSetter) {
      customSetter()
    }
    if (setter) {
      setter.call(obj, newVal)
    } else {
      val = newVal
    }
    childOb = !shallow && observe(newVal)
    dep.notify()
  }
})
}

```

`defineReactive` 函数最开始初始化 `Dep` 对象的实例，接着拿到 `obj` 的属性描述符，然后对子对象递归调用 `observe` 方法，这样就保证了无论 `obj` 的结构多复杂，它的所有子属性也能变成响应式的对象，这样我们访问或修改 `obj` 中一个嵌套较深的属性，也能触发 `getter` 和 `setter`。最后利用 `Object.defineProperty` 去给 `obj` 的属性 `key` 添加 `getter` 和 `setter`。而关于 `getter` 和 `setter` 的具体实现，我们会在之后介绍。

总结

这一节我们介绍了响应式对象，核心就是利用 `Object.defineProperty` 给数据添加了 `getter` 和 `setter`，目的就是为了在我们访问数据以及写数据的时候能自动执行一些逻辑：`getter` 做的事情是依赖收集，`setter` 做的事情是派发更新，那么在接下来的章节我们会重点对这两个过程分析。

依赖收集

通过上一节的分析我们了解 Vue 会把普通对象变成响应式对象，响应式对象 getter 相关的逻辑就是做依赖收集，这一节我们来详细分析这个过程。

我们先来回顾一下 getter 部分的逻辑：

```
export function defineReactive (
  obj: Object,
  key: string,
  val: any,
  customSetter?: ?Function,
  shallow?: boolean
) {
  const dep = new Dep()

  const property = Object.getOwnPropertyDescriptor(obj, key)
  if (property && property.configurable === false) {
    return
  }

  // cater for pre-defined getter/setters
  const getter = property && property.get
  const setter = property && property.set
  if ((!getter || setter) && arguments.length === 2) {
    val = obj[key]
  }

  let childOb = !shallow && observe(val)
  Object.defineProperty(obj, key, {
    enumerable: true,
    configurable: true,
    get: function reactiveGetter () {
      const value = getter ? getter.call(obj) : val
      if (Dep.target) {
        dep.depend()
        if (childOb) {
          childOb.dep.depend()
          if (Array.isArray(value)) {
            dependArray(value)
          }
        }
      }
      return value
    },
    // ...
  })
}
```

这段代码我们只需要关注 2 个地方，一个是 `const dep = new Dep()` 实例化一个 `Dep` 的实例，另一个是在 `get` 函数中通过 `dep.depend` 做依赖收集，这里还有个对 `childobj` 判断的逻辑，我们之后会介绍它的作用。

Dep

`Dep` 是整个 getter 依赖收集的核心，它的定义在 `src/core/observer/dep.js` 中：

```
import type Watcher from './watcher'
import { remove } from '../util/index'

let uid = 0

/**
 * A dep is an observable that can have multiple
 * directives subscribing to it.
 */
export default class Dep {
  static target: ?Watcher;
  id: number;
  subs: Array<Watcher>;

  constructor () {
    this.id = uid++
    this.subs = []
  }

  addSub (sub: Watcher) {
    this.subs.push(sub)
  }

  removeSub (sub: Watcher) {
    remove(this.subs, sub)
  }

  depend () {
    if (Dep.target) {
      Dep.target.addDep(this)
    }
  }

  notify () {
    // stabilize the subscriber list first
    const subs = this.subs.slice()
    for (let i = 0, l = subs.length; i < l; i++) {
      subs[i].update()
    }
  }
}
```

```
// the current target watcher being evaluated.
// this is globally unique because there could be only one
// watcher being evaluated at any time.
Dep.target = null
const targetStack = []

export function pushTarget (_target: ?Watcher) {
  if (Dep.target) targetStack.push(Dep.target)
  Dep.target = _target
}

export function popTarget () {
  Dep.target = targetStack.pop()
}
```

`Dep` 是一个 Class，它定义了一些属性和方法，这里需要特别注意的是它有一个静态属性 `target`，这是一个全局唯一 `Watcher`，这是一个非常巧妙的设计，因为在同一时间只能有一个全局的 `Watcher` 被计算，另外它的自身属性 `subs` 也是 `Watcher` 的数组。

`Dep` 实际上就是对 `Watcher` 的一种管理，`Dep` 脱离 `Watcher` 单独存在是没有意义的，为了完整地讲清楚依赖收集过程，我们有必要看一下 `Watcher` 的一些相关实现，它的定义在 `src/core/observer/watcher.js` 中：

Watcher

```
let uid = 0

/**
 * A watcher parses an expression, collects dependencies,
 * and fires callback when the expression value changes.
 * This is used for both the $watch() api and directives.
 */
export default class Watcher {
  vm: Component;
  expression: string;
  cb: Function;
  id: number;
  deep: boolean;
  user: boolean;
  computed: boolean;
  sync: boolean;
  dirty: boolean;
  active: boolean;
  dep: Dep;
  deps: Array<Dep>;
  newDeps: Array<Dep>;
  depIds: SimpleSet;
```

```

newDepIds: SimpleSet;
before: ?Function;
getter: Function;
value: any;

constructor (
  vm: Component,
  expOrFn: string | Function,
  cb: Function,
  options?: ?Object,
  isRenderWatcher?: boolean
) {
  this.vm = vm
  if (isRenderWatcher) {
    vm._watcher = this
  }
  vm._watchers.push(this)
  // options
  if (options) {
    this.deep = !!options.deep
    this.user = !!options.user
    this.computed = !!options.computed
    this.sync = !!options.sync
    this.before = options.before
  } else {
    this.deep = this.user = this.computed = this.sync = false
  }
  this.cb = cb
  this.id = ++uid // uid for batching
  this.active = true
  this.dirty = this.computed // for computed watchers
  this.deps = []
  this.newDeps = []
  this.depIds = new Set()
  this.newDepIds = new Set()
  this.expression = process.env.NODE_ENV !== 'production'
    ? expOrFn.toString()
    : ''
  // parse expression for getter
  if (typeof expOrFn === 'function') {
    this.getter = expOrFn
  } else {
    this.getter = parsePath(expOrFn)
    if (!this.getter) {
      this.getter = function () {}
    }
    process.env.NODE_ENV !== 'production' && warn(
      `Failed watching path: "${expOrFn}" ` +
      'Watcher only accepts simple dot-delimited paths. ' +
      'For full control, use a function instead.',
      vm
    )
  }
}

```

```

        }
    }
    if (this.computed) {
        this.value = undefined
        this.dep = new Dep()
    } else {
        this.value = this.get()
    }
}

/**
 * Evaluate the getter, and re-collect dependencies.
 */
get () {
    pushTarget(this)
    let value
    const vm = this.vm
    try {
        value = this.getter.call(vm, vm)
    } catch (e) {
        if (this.user) {
            handleError(e, vm, `getter for watcher "${this.expression}"`)
        } else {
            throw e
        }
    }
    finally {
        // "touch" every property so they are all tracked as
        // dependencies for deep watching
        if (this.deep) {
            traverse(value)
        }
        popTarget()
        this.cleanupDeps()
    }
    return value
}

/**
 * Add a dependency to this directive.
 */
addDep (dep: Dep) {
    const id = dep.id
    if (!this.newDepIds.has(id)) {
        this.newDepIds.add(id)
        this.newDeps.push(dep)
        if (!this.depIds.has(id)) {
            dep.addSub(this)
        }
    }
}

```

```

    /**
     * Clean up for dependency collection.
     */
    cleanupDeps () {
        let i = this.deps.length
        while (i--) {
            const dep = this.deps[i]
            if (!this.newDepIds.has(dep.id)) {
                dep.removeSub(this)
            }
        }
        let tmp = this.depIds
        this.depIds = this.newDepIds
        this.newDepIds = tmp
        this.newDepIds.clear()
        tmp = this.deps
        this.deps = this.newDeps
        this.newDeps = tmp
        this.newDeps.length = 0
    }
    // ...
}

```

`Watcher` 是一个 Class，在它的构造函数中，定义了一些和 `Dep` 相关的属性：

```

this.deps = []
this.newDeps = []
this.depIds = new Set()
this.newDepIds = new Set()

```

其中，`this.deps` 和 `this.newDeps` 表示 `Watcher` 实例持有的 `Dep` 实例的数组；而 `this.depIds` 和 `this.newDepIds` 分别代表 `this.deps` 和 `this.newDeps` 的 `id` Set（这个 Set 是 ES6 的数据结构，它的实现再 `src/core/util/env.js` 中）。那么这里为何需要有 2 个 `Dep` 实例数组呢，稍后我们会解释。

`Watcher` 还定义了一些原型的方法，和依赖收集相关的有 `get`、`addDep` 和 `cleanupDeps` 方法，单个介绍它们的实现不方便理解，我会结合整个依赖收集的过程把这几个方法讲清楚。

过程分析

之前我们介绍当对数据对象的访问会触发他们的 `getter` 方法，那么这些对象什么时候被访问呢？还记得之前我们介绍过 Vue 的 `mount` 过程是通过 `mountComponent` 函数，其中有一段比较重要的逻辑，大致如下：

```

updateComponent = () => {
    vm._update(vm._render(), hydrating)
}
new Watcher(vm, updateComponent, noop, {

```

```

before () {
  if (vm._isMounted) {
    callHook(vm, 'beforeUpdate')
  }
},
true /* isRenderWatcher */

```

当我们去实例化一个渲染 `watcher` 的时候，首先进入 `watcher` 的构造函数逻辑，然后会执行它的 `this.get()` 方法，进入 `get` 函数，首先会执行：

```
pushTarget(this)
```

`pushTarget` 的定义在 `src/core/observer/dep.js` 中：

```

export function pushTarget (_target: Watcher) {
  if (Dep.target) targetStack.push(Dep.target)
  Dep.target = _target
}

```

实际上就是把 `Dep.target` 赋值为当前的渲染 `watcher` 并压栈（为了恢复用）。接着又执行了：

```
value = this.getter.call(vm, vm)
```

`this.getter` 对应就是 `updateComponent` 函数，这实际上就是在执行：

```
vm._update(vm._render(), hydrating)
```

它会先执行 `vm._render()` 方法，因为之前分析过这个方法会生成渲染 VNode，并且在这个过程中会对 `vm` 上的数据访问，这个时候就触发了数据对象的 `getter`。

那么每个对象值的 `getter` 都持有一个 `dep`，在触发 `getter` 的时候会调用 `dep.depend()` 方法，也就执行 `Dep.target.addDep(this)`。

刚才我们提到这个时候 `Dep.target` 已经被赋值为渲染 `watcher`，那么就执行到 `addDep` 方法：

```

addDep (dep: Dep) {
  const id = dep.id
  if (!this.newDepIds.has(id)) {
    this.newDepIds.add(id)
    this.newDeps.push(dep)
    if (!this.depIds.has(id)) {
      dep.addSub(this)
    }
  }
}

```

这时候会做一些逻辑判断（保证同一数据不会被添加多次）后执行 `dep.addSub(this)`，那么就会执行 `this.subs.push(sub)`，也就是说把当前的 `watcher` 订阅到这个数据持有的 `dep` 的 `subs` 中，这个目的是为后续数据变化时候能通知到哪些 `subs` 做准备。

所以在 `vm._render()` 过程中，会触发所有数据的 `getter`，这样实际上已经完成了一个依赖收集的过程。那么到这里就结束了么，其实并没有，再完成依赖收集后，还有几个逻辑要执行，首先是：

```
if (this.deep) {
  traverse(value)
}
```

这个是要递归去访问 `value`，触发它所有子项的 `getter`，这个之后会详细讲。接下来执行：

```
popTarget()
```

`popTarget` 的定义在 `src/core/observer/dep.js` 中：

```
Dep.target = targetStack.pop()
```

实际上就是把 `Dep.target` 恢复成上一个状态，因为当前 `vm` 的数据依赖收集已经完成，那么对应的渲染 `Dep.target` 也需要改变。最后执行：

```
this.cleanupDeps()
```

其实很多人都分析过并了解到 Vue 有依赖收集的过程，但我几乎没有看到有人分析依赖清空的过程，其实这是大部分同学会忽视的一点，也是 Vue 考虑特别细的一点。

```
cleanupDeps () {
  let i = this.deps.length
  while (i--) {
    const dep = this.deps[i]
    if (!this.newDepIds.has(dep.id)) {
      dep.removeSub(this)
    }
  }
  let tmp = this.depIds
  this.depIds = this.newDepIds
  this.newDepIds = tmp
  this.newDepIds.clear()
  tmp = this.deps
  this.deps = this.newDeps
  this.newDeps = tmp
  this.newDeps.length = 0
}
```

考虑到 Vue 是数据驱动的，所以每次数据变化都会重新 render，那么 `vm._render()` 方法又会再次执行，并再次触发数据的 getters，所以 `watcher` 在构造函数中会初始化 2 个 `Dep` 实例数组，`newDeps` 表示新添加的 `Dep` 实例数组，而 `deps` 表示上一次添加的 `Dep` 实例数组。

在执行 `cleanupDeps` 函数的时候，会首先遍历 `deps`，移除对 `dep` 的订阅，然后把 `newDepIds` 和 `depIds` 交换，`newDeps` 和 `deps` 交换，并把 `newDepIds` 和 `newDeps` 清空。

那么为什么需要做 `deps` 订阅的移除呢，在添加 `deps` 的订阅过程，已经能通过 `id` 去重避免重复订阅了。

考虑到一种场景，我们的模板会根据 `v-if` 去渲染不同子模板 a 和 b，当我们满足某种条件的时候渲染 a 的时候，会访问到 a 中的数据，这时候我们对 a 使用的数据添加了 getter，做了依赖收集，那么当我们去修改 a 的数据的时候，理应通知到这些订阅者。那么如果我们一旦改变了条件渲染了 b 模板，又会对 b 使用的数据添加了 getter，如果我们没有依赖移除的过程，那么这时候我去修改 a 模板的数据，会通知 a 数据的订阅的回调，这显然是有浪费的。

因此 Vue 设计了在每次添加完新的订阅，会移除掉旧的订阅，这样就保证了在我们刚才的场景中，如果渲染 b 模板的时候去修改 a 模板的数据，a 数据订阅回调已经被移除了，所以不会有任何浪费，真的是非常赞叹 Vue 对一些细节上的处理。

总结

通过这一节的分析，我们对 Vue 数据的依赖收集过程已经有了认识，并且对这其中的一些细节做了分析。收集依赖的目的是为了当这些响应式数据发送变化，触发它们的 `setter` 的时候，能知道应该通知哪些订阅者去做相应的逻辑处理，我们把这个过程叫派发更新，其实 `Watcher` 和 `Dep` 就是一个非常经典的观察者设计模式的实现，下一节我们来详细分析一下派发更新的过程。

派发更新

通过上一节分析我们了解了响应式数据依赖收集过程，收集的目的就是为了当我们修改数据的时候，可以对相关的依赖派发更新，那么这一节我们来详细分析这个过程。

我们先来回顾一下 setter 部分的逻辑：

```
/**
 * Define a reactive property on an Object.
 */
export function defineReactive (
  obj: Object,
  key: string,
  val: any,
  customSetter?: ?Function,
  shallow?: boolean
) {
  const dep = new Dep()

  const property = Object.getOwnPropertyDescriptor(obj, key)
  if (property && property.configurable === false) {
    return
  }

  // cater for pre-defined getter/setters
  const getter = property && property.get
  const setter = property && property.set
  if ((!getter || setter) && arguments.length === 2) {
    val = obj[key]
  }

  let childOb = !shallow && observe(val)
  Object.defineProperty(obj, key, {
    enumerable: true,
    configurable: true,
    // ...
    set: function reactiveSetter (newVal) {
      const value = getter ? getter.call(obj) : val
      /* eslint-disable no-self-compare */
      if (newVal === value || (newVal !== newVal && value !== value)) {
        return
      }
      /* eslint-enable no-self-compare */
      if (process.env.NODE_ENV !== 'production' && customSetter) {
        customSetter()
      }
      if (setter) {
        setter.call(obj, newVal)
      } else {
        dep.add(newVal)
      }
    }
  })
}
```

```

    } else {
      val = newVal
    }
    childOb = !shallow && observe(newVal)
    dep.notify()
  }
})
}

```

setter 的逻辑有 2 个关键的点，一个是 `childOb = !shallow && observe(newVal)`，如果 `shallow` 为 `false` 的情况，会对新设置的值变成一个响应式对象；另一个是 `dep.notify()`，通知所有的订阅者，这是本节的关键，接下来我会带大家完整的分析整个派发更新的过程。

过程分析

当我们在组件中对响应的数据做了修改，就会触发 setter 的逻辑，最后调用 `dep.notify()` 方法，它是 `Dep` 的一个实例方法，定义在 `src/core/observer/dep.js` 中：

```

class Dep {
  // ...
  notify () {
    // stabilize the subscriber list first
    const subs = this.subs.slice()
    for (let i = 0, l = subs.length; i < l; i++) {
      subs[i].update()
    }
  }
}

```

这里的逻辑非常简单，遍历所有的 `subs`，也就是 `watcher` 的实例数组，然后调用每一个 `watcher` 的 `update` 方法，它的定义在 `src/core/observer/watcher.js` 中：

```

class Watcher {
  // ...
  update () {
    /* istanbul ignore else */
    if (this.computed) {
      // A computed property watcher has two modes: lazy and activated.
      // It initializes as lazy by default, and only becomes activated when
      // it is depended on by at least one subscriber, which is typically
      // another computed property or a component's render function.
      if (this.dep.subs.length === 0) {
        // In lazy mode, we don't want to perform computations until necessary,
        // so we simply mark the watcher as dirty. The actual computation is
        // performed just-in-time in this.evaluate() when the computed property
        // is accessed.
        this.dirty = true
      } else {
        // ...
      }
    }
  }
}

```

```

    // In activated mode, we want to proactively perform the computation
    // but only notify our subscribers when the value has indeed changed.
    this.getAndInvoke(() => {
      this.dep.notify()
    })
  }
} else if (this.sync) {
  this.run()
} else {
  queueWatcher(this)
}
}
}
}

```

这里对于 `Watcher` 的不同状态，会执行不同的逻辑，`computed` 和 `sync` 等状态的分析我会之后抽一小节详细介绍，在一般组件数据更新的场景，会走到最后一个 `queueWatcher(this)` 的逻辑，`queueWatcher` 的定义在 `src/core/observer/scheduler.js` 中：

```

const queue: Array<Watcher> = []
let has: { [key: number]: ?true } = {}
let waiting = false
let flushing = false
/** 
 * Push a watcher into the watcher queue.
 * Jobs with duplicate IDs will be skipped unless it's
 * pushed when the queue is being flushed.
 */
export function queueWatcher (watcher: Watcher) {
  const id = watcher.id
  if (has[id] == null) {
    has[id] = true
    if (!flushing) {
      queue.push(watcher)
    } else {
      // if already flushing, splice the watcher based on its id
      // if already past its id, it will be run next immediately.
      let i = queue.length - 1
      while (i > index && queue[i].id > watcher.id) {
        i--
      }
      queue.splice(i + 1, 0, watcher)
    }
    // queue the flush
    if (!waiting) {
      waiting = true
      nextTick(flushSchedulerQueue)
    }
  }
}

```

这里引入了一个队列的概念，这也是 Vue 在做派发更新的时候的一个优化的点，它并不会每次数据改变都触发 `watcher` 的回调，而是把这些 `watcher` 先添加到一个队列里，然后在 `nextTick` 后执行 `flushSchedulerQueue`。

这里有几个细节要注意一下，首先用 `has` 对象保证同一个 `watcher` 只添加一次；接着对 `flushing` 的判断，`else` 部分的逻辑稍后我会讲；最后通过 `wating` 保证对 `nextTick(flushSchedulerQueue)` 的调用逻辑只有一次，另外 `nextTick` 的实现我之后会抽一小节专门去讲，目前就可以理解它是在下一个 tick，也就是异步的去执行 `flushSchedulerQueue`。

接下来我们来看 `flushSchedulerQueue` 的实现，它的定义在 `src/core/observer/scheduler.js` 中。

```
let flushing = false
let index = 0
/**
 * Flush both queues and run the watchers.
 */
function flushSchedulerQueue () {
  flushing = true
  let watcher, id

  // Sort queue before flush.
  // This ensures that:
  // 1. Components are updated from parent to child. (because parent is always
  //     created before the child)
  // 2. A component's user watchers are run before its render watcher (because
  //     user watchers are created before the render watcher)
  // 3. If a component is destroyed during a parent component's watcher run,
  //     its watchers can be skipped.
  queue.sort((a, b) => a.id - b.id)

  // do not cache length because more watchers might be pushed
  // as we run existing watchers
  for (index = 0; index < queue.length; index++) {
    watcher = queue[index]
    if (watcher.before) {
      watcher.before()
    }
    id = watcher.id
    has[id] = null
    watcher.run()
    // in dev build, check and stop circular updates.
    if (process.env.NODE_ENV !== 'production' && has[id] != null) {
      circular[id] = (circular[id] || 0) + 1
      if (circular[id] > MAX_UPDATE_COUNT) {
        warn(
          'You may have an infinite update loop ' +
          watcher.user
          ? `in watcher with expression "${watcher.expression}"` -
          : `in a component render function.`)
      }
    }
  }
}
```

```

        ),
        watcher.vm
    )
break
}
}

// keep copies of post queues before resetting state
const activatedQueue = activatedChildren.slice()
const updatedQueue = queue.slice()

resetSchedulerState()

// call component updated and activated hooks
callActivatedHooks(activatedQueue)
callUpdatedHooks(updatedQueue)

// devtool hook
/* istanbul ignore if */
if (devtools && config.devtools) {
  devtools.emit('flush')
}
}
}

```

这里有几个重要的逻辑要梳理一下，对于一些分支逻辑如 `keep-alive` 组件相关和之前提到过的 `updated` 钩子函数的执行会略过。

- 队列排序

`queue.sort((a, b) => a.id - b.id)` 对队列做了从小到大的排序，这么做主要有以下要确保以下几点：

1.组件的更新由父到子；因为父组件的创建过程是先于子的，所以 `watcher` 的创建也是先父后子，执行顺序也应该保持先父后子。

2.用户的自定义 `watcher` 要优先于渲染 `watcher` 执行；因为用户自定义 `watcher` 是在渲染 `watcher` 之前创建的。

3.如果一个组件在父组件的 `watcher` 执行期间被销毁，那么它对应的 `watcher` 执行都可以被跳过，所以父组件的 `watcher` 应该先执行。

- 队列遍历

在对 `queue` 排序后，接着就是要对它做遍历，拿到对应的 `watcher`，执行 `watcher.run()`。这里需要注意一个细节，在遍历的时候每次都会对 `queue.length` 求值，因为在 `watcher.run()` 的时候，很可能用户会再次添加新的 `watcher`，这样会再次执行到 `queueWatcher`，如下：

```

export function queueWatcher (watcher: Watcher) {
  const id = watcher.id
  if (has[id] == null) {

```

```

has[id] = true
if (!flushing) {
  queue.push(watcher)
} else {
  // if already flushing, splice the watcher based on its id
  // if already past its id, it will be run next immediately.
  let i = queue.length - 1
  while (i > index && queue[i].id > watcher.id) {
    i--
  }
  queue.splice(i + 1, 0, watcher)
}
// ...
}
}

```

可以看到，这时候 `flushing` 为 `true`，就会执行到 `else` 的逻辑，然后就会从后往前找，找到第一个待插入 `watcher` 的 `id` 比当前队列中 `watcher` 的 `id` 大的位置。把 `watcher` 按照 `id` 的插入到队列中，因此 `queue` 的长度发送了变化。

- 状态恢复

这个过程就是执行 `resetSchedulerState` 函数，它的定义在 `src/core/observer/scheduler.js` 中。

```

const queue: Array<Watcher> = []
let has: { [key: number]: ?true } = {}
let circular: { [key: number]: number } = {}
let waiting = false
let flushing = false
let index = 0
/**
 * Reset the scheduler's state.
 */
function resetSchedulerState () {
  index = queue.length = activatedChildren.length = 0
  has = {}
  if (process.env.NODE_ENV !== 'production') {
    circular = {}
  }
  waiting = flushing = false
}

```

逻辑非常简单，就是把这些控制流程状态的一些变量恢复到初始值，把 `watcher` 队列清空。

接下来我们继续分析 `watcher.run()` 的逻辑，它的定义在 `src/core/observer/watcher.js` 中。

```

class Watcher {
  /**

```

```

 * Scheduler job interface.
 * Will be called by the scheduler.
 */
run () {
  if (this.active) {
    this.getAndInvoke(this.cb)
  }
}

getAndInvoke (cb: Function) {
  const value = this.get()
  if (
    value !== this.value ||
    // Deep watchers and watchers on Object/Arrays should fire even
    // when the value is the same, because the value may
    // have mutated.
    isObject(value) ||
    this.deep
  ) {
    // set new value
    const oldValue = this.value
    this.value = value
    this.dirty = false
    if (this.user) {
      try {
        cb.call(this.vm, value, oldValue)
      } catch (e) {
        handleError(e, this.vm, `callback for watcher "${this.expression}"`)
      }
    } else {
      cb.call(this.vm, value, oldValue)
    }
  }
}
}

```

`run` 函数实际上就是执行 `this.getAndInvoke` 方法，并传入 `watcher` 的回调函数。`getAndInvoke` 函数逻辑也很简单，先通过 `this.get()` 得到它当前的值，然后做判断，如果满足新旧值不等、新值是对象类型、`deep` 模式任何一个条件，则执行 `watcher` 的回调，注意回调函数执行的时候会把第一个和第二个参数传入新值 `value` 和旧值 `oldValue`，这就是当我们添加自定义 `watcher` 的时候能在回调函数的参数中拿到新旧值的原因。

那么对于渲染 `watcher` 而言，它在执行 `this.get()` 方法求值的时候，会执行 `getter` 方法：

```

updateComponent = () => {
  vm._update(vm._render(), hydrating)
}

```

所以这就是当我们去修改组件相关的响应式数据的时候，会触发组件重新渲染的原因，接着就会重新执行 `patch` 的过程，但它和首次渲染有所不同，之后我们会花一小节去详细介绍。

总结

通过这一节的分析，我们对 Vue 数据修改派发更新的过程也有了认识，实际上就是当数据发生的时候，触发 `setter` 逻辑，把在依赖过程中订阅的所有观察者，也就是 `watcher`，都触发它们的 `update` 过程，这个过程又利用了队列做了进一步优化，在 `nextTick` 后执行所有 `watcher` 的 `run`，最后执行它们的回调函数。`nextTick` 是 Vue 一个比较核心的实现了，下一节我们来重点分析它的实现。

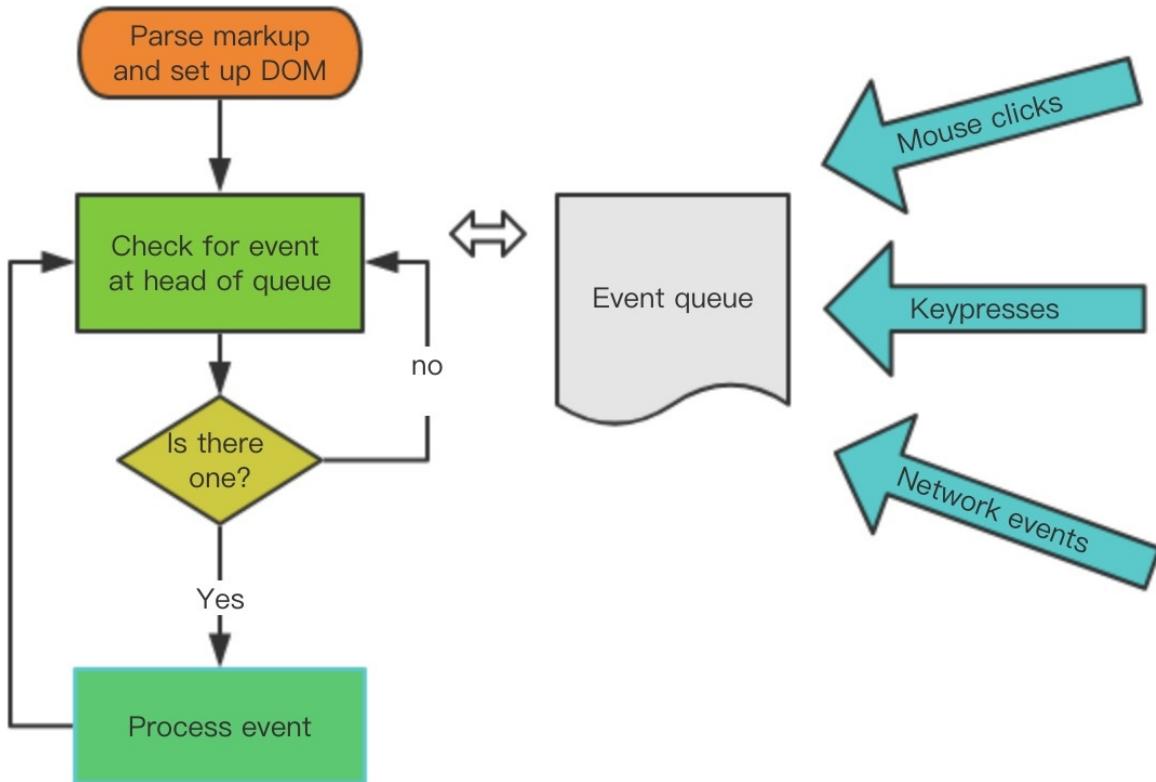
nextTick

`nextTick` 是 Vue 的一个核心实现，在介绍 Vue 的 `nextTick` 之前，为了方便大家理解，我先简单介绍一下 JS 的运行机制。

JS 运行机制

JS 执行是单线程的，它是基于事件循环的。事件循环大致分为以下几个步骤：

- (1) 所有同步任务都在主线程上执行，形成一个执行栈（execution context stack）。
- (2) 主线程之外，还存在一个“任务队列”（task queue）。只要异步任务有了运行结果，就在“任务队列”之中放置一个事件。
- (3) 一旦“执行栈”中的所有同步任务执行完毕，系统就会读取“任务队列”，看看里面有哪些事件。那些对应的异步任务，于是结束等待状态，进入执行栈，开始执行。
- (4) 主线程不断重复上面的第三步。



主线程的执行过程就是一个 tick，而所有的异步结果都是通过“任务队列”来调度被调度。消息队列中存放的是一个个的任务（task）。规范中规定 task 分为两大类，分别是 macro task 和 micro task，并且每个 macro task 结束后，都要清空所有的 micro task。

关于 macro task 和 micro task 的概念，这里不会细讲，简单通过一段代码演示他们的执行顺序：

```

for (macroTask of macroTaskQueue) {
  // 1. Handle current MACRO-TASK
  handleMacroTask();

  // 2. Handle all MICRO-TASK
  for (microTask of microTaskQueue) {
    handleMicroTask(microTask);
  }
}

```

在浏览器环境中，常见的 macro task 有 setTimeout、MessageChannel、postMessage、setImmediate；常见的 micro task 有 MutationObserver 和 Promise.then。

Vue 的实现

在 Vue 源码 2.5+ 后，`nextTick` 的实现单独有一个 JS 文件来维护它，它的源码并不多，总共也就 100 多行。接下来我们来看一下它的实现，在 `src/core/util/next-tick.js` 中：

```

import { noop } from 'shared/util'
import { handleError } from './error'
import { isIOS, isNative } from './env'

const callbacks = []
let pending = false

function flushCallbacks () {
  pending = false
  const copies = callbacks.slice(0)
  callbacks.length = 0
  for (let i = 0; i < copies.length; i++) {
    copies[i]()
  }
}

// Here we have async deferring wrappers using both microtasks and (macro) tasks.
// In < 2.4 we used microtasks everywhere, but there are some scenarios where
// microtasks have too high a priority and fire in between supposedly
// sequential events (e.g. #4521, #6690) or even between bubbling of the same
// event (#6566). However, using (macro) tasks everywhere also has subtle problems
// when state is changed right before repaint (e.g. #6813, out-in transitions).
// Here we use microtask by default, but expose a way to force (macro) task when
// needed (e.g. in event handlers attached by v-on).
let microTimerFunc
let macroTimerFunc
let useMacroTask = false

// Determine (macro) task defer implementation.
// Technically setImmediate should be the ideal choice, but it's only available

```

```

// in IE. The only polyfill that consistently queues the callback after all DOM
// events triggered in the same loop is by using MessageChannel.
/* istanbul ignore if */
if (typeof setImmediate !== 'undefined' && isNative(setImmediate)) {
  macroTimerFunc = () => {
    setImmediate(flushCallbacks)
  }
} else if (typeof MessageChannel !== 'undefined' && (
  isNative(MessageChannel) ||
  // PhantomJS
  MessageChannel.toString() === '[object MessageChannelConstructor]')
)) {
  const channel = new MessageChannel()
  const port = channel.port2
  channel.port1.onmessage = flushCallbacks
  macroTimerFunc = () => {
    port.postMessage(1)
  }
} else {
  /* istanbul ignore next */
  macroTimerFunc = () => {
    setTimeout(flushCallbacks, 0)
  }
}

// Determine microtask defer implementation.
/* istanbul ignore next, $flow-disable-line */
if (typeof Promise !== 'undefined' && isNative(Promise)) {
  const p = Promise.resolve()
  microTimerFunc = () => {
    p.then(flushCallbacks)
    // in problematic UIWebViews, Promise.then doesn't completely break, but
    // it can get stuck in a weird state where callbacks are pushed into the
    // microtask queue but the queue isn't being flushed, until the browser
    // needs to do some other work, e.g. handle a timer. Therefore we can
    // "force" the microtask queue to be flushed by adding an empty timer.
    if (isIOS) setTimeout(noop)
  }
} else {
  // fallback to macro
  microTimerFunc = macroTimerFunc
}

/**
 * Wrap a function so that if any code inside triggers state change,
 * the changes are queued using a (macro) task instead of a microtask.
 */
export function withMacroTask (fn: Function): Function {
  return fn._withTask || (fn._withTask = function () {
    useMacroTask = true
    const res = fn.apply(null, arguments)
  })
}

```

```

useMacroTask = false
return res
})
}

export function nextTick (cb?: Function, ctx?: Object) {
let _resolve
callbacks.push(() => {
if (cb) {
try {
cb.call(ctx)
} catch (e) {
handleError(e, ctx, 'nextTick')
}
} else if (_resolve) {
_resolve(ctx)
}
})
if (!pending) {
pending = true
if (useMacroTask) {
macroTimerFunc()
} else {
microTimerFunc()
}
}
// $flow-disable-line
if (!cb && typeof Promise !== 'undefined') {
return new Promise(resolve => {
_resolve = resolve
})
}
}
}

```

`next-tick.js` 申明了 `microTimerFunc` 和 `macroTimerFunc` 2个变量，它们分别对应的是 micro task 的函数和 macro task 的函数。对于 macro task 的实现，优先检测是否支持原生 `setImmediate`，这是一个高版本 IE 和 Edge 才支持的特性，不支持的话再去检测是否支持原生的 `MessageChannel`，如果也不支持的话就会降级为 `setTimeout 0`；而对于 micro task 的实现，则检测浏览器是否原生支持 `Promise`，不支持的话直接指向 macro task 的实现。

`next-tick.js` 对外暴露了 2 个函数，先来看 `nextTick`，这就是我们在上一节执行 `nextTick(flushSchedulerQueue)` 所用到的函数。它的逻辑也很简单，把传入的回调函数 `cb` 压入 `callbacks` 数组，最后一次性地根据 `useMacroTask` 条件执行 `macroTimerFunc` 或者是 `microTimerFunc`，而它们都会在下一个 tick 执行 `flushCallbacks`，`flushCallbacks` 的逻辑非常简单，对 `callbacks` 遍历，然后执行相应的回调函数。

这里使用 `callbacks` 而不是直接在 `nextTick` 中执行回调函数的原因是保证在同一个 tick 内多次执行 `nextTick`，不会开启多个异步任务，而把这些异步任务都压成一个同步任务，在下一个 tick 执行完毕。

`nextTick` 函数最后还有一段逻辑：

```
if (!cb && typeof Promise !== 'undefined') {
  return new Promise(resolve => {
    _resolve = resolve
  })
}
```

这是当 `nextTick` 不传 `cb` 参数的时候，提供一个 `Promise` 化的调用，比如：

```
nextTick().then(() => {})
```

当 `_resolve` 函数执行，就会跳到 `then` 的逻辑中。

`next-tick.js` 还对外暴露了 `withMacroTask` 函数，它是对函数做一层包装，确保函数执行过程中对数据任意的修改，触发变化执行 `nextTick` 的时候强制走 `macroTimerFunc`。比如对于一些 DOM 交互事件，如 `v-on` 绑定的事件回调函数的处理，会强制走 macro task。

总结

通过这一节对 `nextTick` 的分析，并结合上一节的 `setter` 分析，我们了解到数据的变化到 DOM 的重新渲染是一个异步过程，发生在下一个 tick。这就是我们平时在开发的过程中，比如从服务端接口去获取数据的时候，数据做了修改，如果我们的某些方法去依赖了数据修改后的 DOM 变化，我们就必须在 `nextTick` 后执行。比如下面的伪代码：

```
getData(res).then(()=>{
  this.xxx = res.data
  this.$nextTick(() => {
    // 这里我们可以获取变化后的 DOM
  })
})
```

Vue.js 提供了 2 种调用 `nextTick` 的方式，一种是全局 API `Vue.nextTick`，一种是实例上的方法 `vm.$nextTick`，无论我们使用哪一种，最后都是调用 `next-tick.js` 中实现的 `nextTick` 方法。

检测变化的注意事项

通过前面几节的分析，我们对响应式数据对象以及它的 getter 和 setter 部分做了了解，但是对于一些特殊情况是需要注意的，接下来我们就从源码的角度来看 Vue 是如何处理这些特殊情况的。

对象添加属性

对于使用 `Object.defineProperty` 实现响应式的对象，当我们去给这个对象添加一个新的属性的时候，是不能够触发它的 setter 的，比如：

```
var vm = new Vue({
  data: {
    a: 1
  }
})
// vm.b 是非响应的
vm.b = 2
```

但是添加新属性的场景我们在平时开发中会经常遇到，那么 Vue 为了解决这个问题，定义了一个全局 API `Vue.set` 方法，它在 `src/core/global-api/index.js` 中初始化：

```
Vue.set = set
```

这个 `set` 方法的定义在 `src/core/observer/index.js` 中：

```
/**
 * Set a property on an object. Adds the new property and
 * triggers change notification if the property doesn't
 * already exist.
 */
export function set (target: Array<any> | Object, key: any, val: any): any {
  if (process.env.NODE_ENV !== 'production' &&
    (isUndef(target) || isPrimitive(target)))
  ) {
    warn(`Cannot set reactive property on undefined, null, or primitive value: ${target}`)
  }
  if (Array.isArray(target) && isValidArrayIndex(key)) {
    target.length = Math.max(target.length, key)
    target.splice(key, 1, val)
    return val
  }
  if (key in target && !(key in Object.prototype)) {
    target[key] = val
    return val
  }
}
```

```

    }
    const ob = (target: any).__ob__
    if (target._isVue || (ob && ob.vmCount)) {
        process.env.NODE_ENV !== 'production' && warn(
            'Avoid adding reactive properties to a Vue instance or its root $data ' +
            'at runtime - declare it upfront in the data option.'
        )
        return val
    }
    if (!ob) {
        target[key] = val
        return val
    }
    defineReactive(ob.value, key, val)
    ob.dep.notify()
    return val
}

```

`set` 方法接收 3 个参数，`target` 可能是数组或者是普通对象，`key` 代表的是数组的下标或者是对象的键值，`val` 代表添加的值。首先判断如果 `target` 是数组且 `key` 是一个合法的下标，则之前通过 `splice` 去添加进数组然后返回，这里的 `splice` 其实已经不仅仅是原生数组的 `splice` 了，稍后我会详细介绍数组的逻辑。接着又判断 `key` 已经存在于 `target` 中，则直接赋值返回，因为这样的变化是可以观测到了。接着再获取到 `target.__ob__` 并赋值给 `ob`，之前分析过它是在 `Observer` 的构造函数执行的时候初始化的，表示 `Observer` 的一个实例，如果它不存在，则说明 `target` 不是一个响应式的对象，则直接赋值并返回。最后通过 `defineReactive(ob.value, key, val)` 把新添加的属性变成响应式对象，然后再通过 `ob.dep.notify()` 手动的触发依赖通知，还记得我们在给对象添加 `getter` 的时候有这么一段逻辑：

```

export function defineReactive (
    obj: Object,
    key: string,
    val: any,
    customSetter?: ?Function,
    shallow?: boolean
) {
    // ...
    let childOb = !shallow && observe(val)
    Object.defineProperty(obj, key, {
        enumerable: true,
        configurable: true,
        get: function reactiveGetter () {
            const value = getter ? getter.call(obj) : val
            if (Dep.target) {
                dep.depend()
                if (childOb) {
                    childOb.dep.depend()
                    if (Array.isArray(value)) {
                        dependArray(value)
                    }
                }
            }
        }
    })
}

```

```

        }
    }
    return value
},
// ...
})
}

```

在 getter 过程中判断了 `childOb`，并调用了 `childOb.dep.depend()` 收集了依赖，这就是为什么执行 `Vue.set` 的时候通过 `ob.dep.notify()` 能够通知到 `watcher`，从而让添加新的属性到对象也可以检测到变化。这里如果 `value` 是个数组，那么就通过 `dependArray` 把数组每个元素也去做依赖收集。

数组

接着说一下数组的情况，`Vue` 也是不能检测到以下变动的数组：

1.当你利用索引直接设置一个项时，例如：`vm.items[indexOfItem] = newValue`

2.当你修改数组的长度时，例如：`vm.items.length = newLength`

对于第一种情况，可以使用：`Vue.set(example1.items, indexOfItem, newValue)`；而对于第二种情况，可以使用 `vm.items.splice(newLength)`。

我们刚才也分析到，对于 `Vue.set` 的实现，当 `target` 是数组的时候，也是通过 `target.splice(key, 1, val)` 来添加的，那么这里的 `splice` 到底有什么黑魔法，能让添加的对象变成响应式的呢。

其实之前我们也分析过，在通过 `observe` 方法去观察对象的时候会实例化 `observer`，在它的构造函数中是专门为数组做了处理，它的定义在 `src/core/observer/index.js` 中。

```

export class Observer {
  constructor (value: any) {
    this.value = value
    this.dep = new Dep()
    this.vmCount = 0
    def(value, '__ob__', this)
    if (Array.isArray(value)) {
      const augment = hasProto
        ? protoAugment
        : copyAugment
      augment(value, arrayMethods, arrayKeys)
      this.observeArray(value)
    } else {
      // ...
    }
  }
}

```

这里我们只需要关注 `value` 是 `Array` 的情况，首先获取 `augment`，这里的 `hasProto` 实际上就是判断对象中是否存在 `__proto__`，如果存在则 `augment` 指向 `protoAugment`，否则指向 `copyAugment`，来看一下这两个函数的定义：

```
/**
 * Augment an target Object or Array by intercepting
 * the prototype chain using __proto__
 */
function protoAugment (target, src: Object, keys: any) {
    /* eslint-disable no-proto */
    target.__proto__ = src
    /* eslint-enable no-proto */
}

/**
 * Augment an target Object or Array by defining
 * hidden properties.
 */
/* istanbul ignore next */
function copyAugment (target: Object, src: Object, keys: Array<string>) {
    for (let i = 0, l = keys.length; i < l; i++) {
        const key = keys[i]
        def(target, key, src[key])
    }
}
```

`protoAugment` 方法是直接把 `target.__proto__` 原型直接修改为 `src`，而 `copyAugment` 方法是遍历 `keys`，通过 `def`，也就是 `Object.defineProperty` 去定义它自身的属性值。对于大部分现代浏览器都会走到 `protoAugment`，那么它实际上就把 `value` 的原型指向了 `arrayMethods`，`arrayMethods` 的定义在 `src/core/observer/array.js` 中：

```
import { def } from '../util/index'

const arrayProto = Array.prototype
export const arrayMethods = Object.create(arrayProto)

const methodsToPatch = [
    'push',
    'pop',
    'shift',
    'unshift',
    'splice',
    'sort',
    'reverse'
]

/**
 * Intercept mutating methods and emit events
 */
```

```

methodsToPatch.forEach(function (method) {
  // cache original method
  const original = arrayProto[method]
  def(arrayMethods, method, function mutator (...args) {
    const result = original.apply(this, args)
    const ob = this.__ob__
    let inserted
    switch (method) {
      case 'push':
      case 'unshift':
        inserted = args
        break
      case 'splice':
        inserted = args.slice(2)
        break
    }
    if (inserted) ob.observeArray(inserted)
    // notify change
    ob.dep.notify()
    return result
  })
})
}

```

可以看到，`arrayMethods` 首先继承了 `Array`，然后对数组中所有能改变数组自身的方法，如 `push`、`pop` 等这些方法进行重写。重写后的方法会先执行它们本身原有的逻辑，并对能增加数组长度的 3 个方法 `push`、`unshift`、`splice` 方法做了判断，获取到插入的值，然后把新添加的值变成一个响应式对象，并且再调用 `ob.dep.notify()` 手动触发依赖通知，这就很好地解释了之前的示例中调用 `vm.items.splice(newLength)` 方法可以检测到变化。

总结

通过这一节的分析，我们对响应式对象又有了更全面的认识，如果在实际工作中遇到了这些特殊情况，我们就可以知道如何把它们也变成响应式的对象。其实对于对象属性的删除也会用同样的问题，Vue 同样提供了 `Vue.del` 的全局 API，它的实现和 `Vue.set` 大相径庭，甚至还要更简单一些，这里我就不去分析了，感兴趣的同学可以自行去了解。

计算属性 VS 值属性

Vue 的组件对象支持了计算属性 `computed` 和值属性 `watch` 2 个选项，很多同学不了解什么时候该用 `computed` 什么时候该用 `watch`。先不回答这个问题，我们接下来从源码实现的角度来分析它们两者有什么区别。

computed

计算属性的初始化是发生在 Vue 实例初始化阶段的 `initState` 函数中，执行了 `if (opts.computed) initComputed(vm, opts.computed)`，`initComputed` 的定义在 `src/core/instance/state.js` 中：

```
const computedWatcherOptions = { computed: true }
function initComputed (vm: Component, computed: Object) {
  // $flow-disable-line
  const watchers = vm._computedWatchers = Object.create(null)
  // computed properties are just getters during SSR
  const isSSR = isServerRendering()

  for (const key in computed) {
    const userDef = computed[key]
    const getter = typeof userDef === 'function' ? userDef : userDef.get
    if (process.env.NODE_ENV !== 'production' && getter == null) {
      warn(
        `Getter is missing for computed property "${key}".`,
        vm
      )
    }

    if (!isSSR) {
      // create internal watcher for the computed property.
      watchers[key] = new Watcher(
        vm,
        getter || noop,
        noop,
        computedWatcherOptions
      )
    }
  }

  // component-defined computed properties are already defined on the
  // component prototype. We only need to define computed properties defined
  // at instantiation here.
  if (!(key in vm)) {
    defineComputed(vm, key, userDef)
  } else if (process.env.NODE_ENV !== 'production') {
    if (key in vm.$data) {
      warn(`The computed property "${key}" is already defined in data.`, vm)
    }
  }
}
```

```

        } else if (vm.$options.props && key in vm.$options.props) {
            warn(`The computed property "${key}" is already defined as a prop.`,
                vm)
        }
    }
}

```

函数首先创建 `vm._computedWatchers` 为一个空对象，接着对 `computed` 对象做遍历，拿到计算属性的每一个 `userDef`，然后尝试获取这个 `userDef` 对应的 `getter` 函数，拿不到则在开发环境下报警告。接下来为每一个 `getter` 创建一个 `watcher`，这个 `watcher` 和渲染 `watcher` 有一点很大的不同，它是一个 `computed watcher`，因为 `const computedWatcherOptions = { computed: true }`。`computed watcher` 和普通 `watcher` 的差别我稍后会介绍。最后对判断如果 `key` 不是 `vm` 的属性，则调用 `defineComputed(vm, key, userDef)`，否则判断计算属性对于的 `key` 是否已经被 `data` 或者 `prop` 所占用，如果是的话则在开发环境报相应的警告。

那么接下来需要重点关注 `defineComputed` 的实现：

```

export function defineComputed (
    target: any,
    key: string,
    userDef: Object | Function
) {
    const shouldCache = !isServerRendering()
    if (typeof userDef === 'function') {
        sharedPropertyDefinition.get = shouldCache
            ? createComputedGetter(key)
            : userDef
        sharedPropertyDefinition.set = noop
    } else {
        sharedPropertyDefinition.get = userDef.get
            ? shouldCache && userDef.cache !== false
                ? createComputedGetter(key)
                : userDef.get
            : noop
        sharedPropertyDefinition.set = userDef.set
            ? userDef.set
            : noop
    }
    if (process.env.NODE_ENV !== 'production' &&
        sharedPropertyDefinition.set === noop) {
        sharedPropertyDefinition.set = function () {
            warn(
                `Computed property "${key}" was assigned to but it has no setter.`,
                this
            )
        }
    }
    Object.defineProperty(target, key, sharedPropertyDefinition)
}

```

这段逻辑很简单，其实就是利用 `Object.defineProperty` 给计算属性对应的 `key` 值添加 `getter` 和 `setter`，`setter` 通常是计算属性是一个对象，并且拥有 `set` 方法的时候才有，否则是一个空函数。在平时的开发场景中，计算属性有 `setter` 的情况比较少，我们重点关注一下 `getter` 部分，缓存的配置也先忽略，最终 `getter` 对应的是 `createComputedGetter(key)` 的返回值，来看一下它的定义：

```
function createComputedGetter (key) {
  return function computedGetter () {
    const watcher = this._computedWatchers && this._computedWatchers[key]
    if (watcher) {
      watcher.depend()
      return watcher.evaluate()
    }
  }
}
```

`createComputedGetter` 返回一个函数 `computedGetter`，它就是计算属性对应的 `getter`。

整个计算属性的初始化过程到此结束，我们知道计算属性是一个 `computed watcher`，它和普通的 `watcher` 有什么区别呢，为了更加直观，接下来我们来通过一个例子来分析 `computed watcher` 的实现。

```
var vm = new Vue({
  data: {
    firstName: 'Foo',
    lastName: 'Bar'
  },
  computed: {
    fullName: function () {
      return this.firstName + ' ' + this.lastName
    }
  }
})
```

当初始化这个 `computed watcher` 实例的时候，构造函数部分逻辑稍有不同：

```
constructor (
  vm: Component,
  expOrFn: string | Function,
  cb: Function,
  options?: ?Object,
  isRenderWatcher?: boolean
) {
  // ...
  if (this.computed) {
    this.value = undefined
    this.dep = new Dep()
  } else {
    this.value = this.get()
```

```

    }
}

```

可以发现 `computed watcher` 会并不会立刻求值，同时持有一个 `dep` 实例。

然后当我们的 `render` 函数执行访问到 `this.fullName` 的时候，就触发了计算属性的 `getter`，它会拿到计算属性对应的 `watcher`，然后执行 `watcher.depend()`，来看一下它的定义：

```

/**
 * Depend on this watcher. Only for computed property watchers.
 */
depend () {
  if (this.dep && Dep.target) {
    this.dep.depend()
  }
}

```

注意，这时候的 `Dep.target` 是渲染 `watcher`，所以 `this.dep.depend()` 相当于渲染 `watcher` 订阅了这个 `computed watcher` 的变化。

然后再执行 `watcher.evaluate()` 去求值，来看一下它的定义：

```

/**
 * Evaluate and return the value of the watcher.
 * This only gets called for computed property watchers.
 */
evaluate () {
  if (this.dirty) {
    this.value = this.get()
    this.dirty = false
  }
  return this.value
}

```

`evaluate` 的逻辑非常简单，判断 `this.dirty`，如果为 `true` 则通过 `this.get()` 求值，然后把 `this.dirty` 设置为 `false`。在求值过程中，会执行 `value = this.getter.call(vm, vm)`，这实际上就是执行了计算属性定义的 `getter` 函数，在我们这个例子就是执行了 `return this.firstName + ' ' + this.lastName`。

这里需要特别注意的是，由于 `this.firstName` 和 `this.lastName` 都是响应式对象，这里会触发它们的 `getter`，根据我们之前的分析，它们会把自身持有的 `dep` 添加到当前正在计算的 `watcher` 中，这个时候 `Dep.target` 就是这个 `computed watcher`。

最后通过 `return this.value` 拿到计算属性对应的值。我们知道了计算属性的求值过程，那么接下来看一下它依赖的数据变化后的逻辑。

一旦我们对计算属性依赖的数据做修改，则会触发 `setter` 过程，通知所有订阅它变化的 `watcher` 更新，执行 `watcher.update()` 方法：

```

/* istanbul ignore else */
if (this.computed) {
    // A computed property watcher has two modes: lazy and activated.
    // It initializes as lazy by default, and only becomes activated when
    // it is depended on by at least one subscriber, which is typically
    // another computed property or a component's render function.
    if (this.dep.subs.length === 0) {
        // In lazy mode, we don't want to perform computations until necessary,
        // so we simply mark the watcher as dirty. The actual computation is
        // performed just-in-time in this.evaluate() when the computed property
        // is accessed.
        this.dirty = true
    } else {
        // In activated mode, we want to proactively perform the computation
        // but only notify our subscribers when the value has indeed changed.
        this.getAndInvoke(() => {
            this.dep.notify()
        })
    }
} else if (this.sync) {
    this.run()
} else {
    queueWatcher(this)
}

```

那么对于计算属性这样的 `computed watcher`，它实际上是有 2 种模式，`lazy` 和 `active`。如果 `this.dep.subs.length === 0` 成立，则说明没有人去订阅这个 `computed watcher` 的变化，仅仅把 `this.dirty = true`，只有当下次再访问这个计算属性的时候才会重新求值。在我们的场景下，渲染 `watcher` 订阅了这个 `computed watcher` 的变化，那么它会执行：

```

this.getAndInvoke(() => {
    this.dep.notify()
})

getAndInvoke (cb: Function) {
    const value = this.get()
    if (
        value !== this.value ||
        // Deep watchers and watchers on Object/Arrays should fire even
        // when the value is the same, because the value may
        // have mutated.
        isObject(value) ||
        this.deep
    ) {
        // set new value
        const oldValue = this.value
        this.value = value
        this.dirty = false
        if (this.user) {

```

```

    try {
      cb.call(this.vm, value, oldValue)
    } catch (e) {
      handleError(e, this.vm, `callback for watcher "${this.expression}"`)
    }
  } else {
    cb.call(this.vm, value, oldValue)
  }
}
}

```

`getAndInvoke` 函数会重新计算，然后对比新旧值，如果变化了则执行回调函数，那么这里这个回调函数是 `this.dep.notify()`，在我们这个场景下就是触发了渲染 `watcher` 重新渲染。

通过以上的分析，我们知道计算属性本质上就是一个 `computed watcher`，也了解了它的创建过程和被访问触发 `getter` 以及依赖更新的过程，其实这是最新的计算属性的实现，之所以这么设计是因为 Vue 想确保不仅仅是计算属性依赖的值发生变化，而是当计算属性最终计算的值发生变花才会触发渲染 `watcher` 重新渲染，本质上是一种优化。

接下来我们来分析一下倾听属性 `watch` 是怎么实现的。

watch

倾听属性的初始化也是发生在 Vue 的实例初始化阶段的 `initState` 函数中，在 `computed` 初始化之后，执行了：

```

if (opts.watch && opts.watch !== nativeWatch) {
  initWatch(vm, opts.watch)
}

```

来看一下 `initWatch` 的实现，它的定义在 `src/core/instance/state.js` 中：

```

function initWatch (vm: Component, watch: Object) {
  for (const key in watch) {
    const handler = watch[key]
    if (Array.isArray(handler)) {
      for (let i = 0; i < handler.length; i++) {
        createWatcher(vm, key, handler[i])
      }
    } else {
      createWatcher(vm, key, handler)
    }
  }
}

```

这里就是对 `watch` 对象做遍历，拿到每一个 `handler`，因为 Vue 是支持 `watch` 的同一个 `key` 对应多个 `handler`，所以如果 `handler` 是一个数组，则遍历这个数组，调用 `createWatcher` 方法，否则直接调用 `createWatcher`：

```
function createWatcher (
  vm: Component,
  expOrFn: string | Function,
  handler: any,
  options?: Object
) {
  if (isPlainObject(handler)) {
    options = handler
    handler = handler.handler
  }
  if (typeof handler === 'string') {
    handler = vm[handler]
  }
  return vm.$watch(expOrFn, handler, options)
}
```

这里的逻辑也很简单，首先对 `handler` 的类型做判断，拿到它最终的回调函数，最后调用 `vm.$watch(keyOrFn, handler, options)` 函数，`$watch` 是 Vue 原型上的方法，它是在执行 `stateMixin` 的时候定义的：

```
Vue.prototype.$watch = function (
  expOrFn: string | Function,
  cb: any,
  options?: Object
): Function {
  const vm: Component = this
  if (isPlainObject(cb)) {
    return createWatcher(vm, expOrFn, cb, options)
  }
  options = options || {}
  options.user = true
  const watcher = new Watcher(vm, expOrFn, cb, options)
  if (options.immediate) {
    cb.call(vm, watcher.value)
  }
  return function unwatchFn () {
    watcher.teardown()
  }
}
```

也就是说，值属性 `watch` 最终会调用 `$watch` 方法，这个方法首先判断 `cb` 如果是一个对象，则调用 `createWatcher` 方法，这是因为 `$watch` 方法是用户可以直接调用的，它可以传递一个对象，也可以传递函数。接着执行 `const watcher = new Watcher(vm, expOrFn, cb, options)` 实例化了一个 `watcher`，这里需要注意一点这是一个 `user watcher`，因为 `options.user = true`。

通过实例化 `watcher` 的方式，一旦我们 `watch` 的数据发生变化，它最终会执行 `watcher` 的 `run` 方法，执行回调函数 `cb`，并且如果我们设置了 `immediate` 为 `true`，则直接会执行回调函数 `cb`。最后返回了一个 `unwatchFn` 方法，它会调用 `teardown` 方法去移除这个 `watcher`。

所以本质上倾听属性也是基于 `watcher` 实现的，它是一个 `user watcher`。其实 `watcher` 支持了不同的类型，下面我们梳理一下它有哪些类型以及它们的作用。

Watcher options

`Watcher` 的构造函数对 `options` 做了处理，代码如下：

```
if (options) {
  this.deep = !!options.deep
  this.user = !!options.user
  this.computed = !!options.computed
  this.sync = !!options.sync
  // ...
} else {
  this.deep = this.user = this.computed = this.sync = false
}
```

所以 `watcher` 总共有 4 种类型，我们来一一分析它们，看看不同的类型执行的逻辑有哪些差别。

deep watcher

通常，如果我们想对一下对象做深度观测的时候，需要设置这个属性为 `true`，考虑到这种情况：

```
var vm = new Vue({
  data() {
    a: {
      b: 1
    }
  },
  watch: {
    a: {
      handler(newVal) {
        console.log(newVal)
      }
    }
  }
})
vm.a.b = 2
```

这个时候是不会 `log` 任何数据的，因为我们是 `watch` 了 `a` 对象，只触发了 `a` 的 `getter`，并没有触发 `a.b` 的 `getter`，所以并没有订阅它的变化，导致我们对 `vm.a.b = 2` 赋值的时候，虽然触发了 `setter`，但没有可知的对像，所以也并不会触发 `watch` 的回调函数了。

而我们只需要对代码做稍稍修改，就可以观测到这个变化了

```
watch: {
  a: {
    deep: true,
    handler(newVal) {
      console.log(newVal)
    }
  }
}
```

这样就创建了一个 `deep watcher` 了，在 `watcher` 执行 `get` 求值的过程中有一段逻辑：

```
get() {
  let value = this.getter.call(vm, vm)
  // ...
  if (this.deep) {
    traverse(value)
  }
}
```

在对 `watch` 的表达式或者函数求值后，会调用 `traverse` 函数，它的定义在 `src/core/observer/traverse.js` 中：

```
import { _Set as Set, isObject } from '../util/index'
import type { SimpleSet } from '../util/index'
import VNode from '../vdom/vnode'

const seenObjects = new Set()

/**
 * Recursively traverse an object to evoke all converted
 * getters, so that every nested property inside the object
 * is collected as a "deep" dependency.
 */
export function traverse (val: any) {
  _traverse(val, seenObjects)
  seenObjects.clear()
}

function _traverse (val: any, seen: SimpleSet) {
  let i, keys
  const isArray = Array.isArray(val)
  if ((!isArray && !isObject(val)) || Object.isFrozen(val) || val instanceof VNode) {
    return
  }
  if (val.__ob__) {
    const depId = val.__ob__.dep.id
    if (seen.has(depId)) {

```

```

        return
    }
    seen.add(depId)
}
if (isA) {
    i = val.length
    while (i--) _traverse(val[i], seen)
} else {
    keys = Object.keys(val)
    i = keys.length
    while (i--) _traverse(val[keys[i]], seen)
}
}
}

```

`traverse` 的逻辑也很简单，它实际上就是对一个对象做深层递归遍历，因为遍历过程中就是对一个子对象的访问，会触发它们的 `getter` 过程，这样就可以收集到依赖，也就是订阅它们变化的 `watcher`，这个函数实现还有一个小的优化，遍历过程中会把子响应式对象通过它们的 `dep id` 记录到 `seenObjects`，避免以后重复访问。

那么在执行了 `traverse` 后，我们再对 `watch` 的对象内部任何一个值做修改，也会调用 `watcher` 的回调函数了。

对 `deep watcher` 的理解非常重要，今后工作中如果大家观测了一个复杂对象，并且会改变对象内部深层某个值的时候也希望触发回调，一定要设置 `deep` 为 `true`，但是因为设置了 `deep` 后会执行 `traverse` 函数，会有一定的性能开销，所以一定要根据应用场景权衡是否要开启这个配置。

user watcher

前面我们分析过，通过 `vm.$watch` 创建的 `watcher` 是一个 `user watcher`，其实它的功能很简单，在对 `watcher` 求值以及在执行回调函数的时候，会处理一下错误，如下：

```

get() {
    if (this.user) {
        handleError(e, vm, `getter for watcher "${this.expression}"`)
    } else {
        throw e
    }
},
getAndInvoke() {
    // ...
    if (this.user) {
        try {
            this.cb.call(this.vm, value, oldValue)
        } catch (e) {
            handleError(e, this.vm, `callback for watcher "${this.expression}"`)
        }
    } else {
        this.cb.call(this.vm, value, oldValue)
    }
}

```

```
}
```

`handleError` 在 Vue 中是一个错误捕获并且暴露给用户的一个利器。

computed watcher

`computed watcher` 几乎就是为计算属性量身定制的，我们刚才已经对它做了详细的分析，这里不再赘述了。

sync watcher

在我们之前对 `setter` 的分析过程中知道，当响应式数据发送变化后，触发了 `watcher.update()`，只是把这个 `watcher` 推送到一个队列中，在 `nextTick` 后才会真正执行 `watcher` 的回调函数。而一旦我们设置了 `sync`，就可以在当前 `Tick` 中同步执行 `watcher` 的回调函数。

```
update () {
  if (this.computed) {
    // ...
  } else if (this.sync) {
    this.run()
  } else {
    queueWatcher(this)
  }
}
```

只有当我们需要 `watch` 的值的变化到执行 `watcher` 的回调函数是一个同步过程的时候才会去设置该属性为 `true`。

总结

通过这一小节的分析我们对计算属性和倾听属性的实现有了深入的了解，计算属性本质上是 `computed watcher`，而倾听属性本质上是 `user watcher`。就应用场景而言，计算属性适合用在模板渲染中，某个值是依赖了其它的响应式对象甚至是计算属性计算而来；而倾听属性适用于观测某个值的变化去完成一段复杂的业务逻辑。

同时我们又了解了 `watcher` 的 4 个 `options`，通常我们会在创建 `user watcher` 的时候配置 `deep` 和 `sync`，可以根据不同的场景做相应的配置。

组件更新

在组件化章节，我们介绍了 Vue 的组件化实现过程，不过我们只讲了 Vue 组件的创建过程，并没有涉及到组件数据发生变化，更新组件的过程。而通过我们这一章对数据响应式原理的分析，了解到当数据发生变化的时候，会触发渲染 `watcher` 的回调函数，进而执行组件的更新过程，接下来我们来详细分析这一过程。

```
updateComponent = () => {
  vm._update(vm._render(), hydrating)
}

new Watcher(vm, updateComponent, noop, {
  before () {
    if (vm._isMounted) {
      callHook(vm, 'beforeUpdate')
    }
  }
}, true /* isRenderWatcher */)
```

组件的更新还是调用了 `vm._update` 方法，我们再回顾一下这个方法，它的定义在 `src/core/instance/lifecycle.js` 中：

```
Vue.prototype._update = function (vnode: VNode, hydrating?: boolean) {
  const vm: Component = this
  // ...
  const prevVnode = vm._vnode
  if (!prevVnode) {
    // initial render
    vm.$el = vm.__patch__(vm.$el, vnode, hydrating, false /* removeOnly */)
  } else {
    // updates
    vm.$el = vm.__patch__(prevVnode, vnode)
  }
  // ...
}
```

组件更新的过程，会执行 `vm.$el = vm.__patch__(prevVnode, vnode)`，它仍然会调用 `patch` 函数，在 `src/core/vdom/patch.js` 中定义：

```
return function patch (oldVnode, vnode, hydrating, removeOnly) {
  if (isUndef(vnode)) {
    if (isDef(oldVnode)) invokeDestroyHook(oldVnode)
    return
  }

  let isInitialPatch = false
  const insertedVnodeQueue = []
```

```

if (isUndef(oldVnode)) {
  // empty mount (likely as component), create new root element
  isInitialPatch = true
  createElm(vnode, insertedVnodeQueue)
} else {
  const isRealElement = isDef(oldVnode.nodeType)
  if (!isRealElement && sameVnode(oldVnode, vnode)) {
    // patch existing root node
    patchVnode(oldVnode, vnode, insertedVnodeQueue, removeOnly)
  } else {
    if (isRealElement) {
      // ...
    }

    // replacing existing element
    const oldElm = oldVnode.elm
    const parentElm = nodeOps.parentNode(oldElm)

    // create new node
    createElm(
      vnode,
      insertedVnodeQueue,
      // extremely rare edge case: do not insert if old element is in a
      // leaving transition. Only happens when combining transition +
      // keep-alive + HOCs. (#4590)
      oldElm._leaveCb ? null : parentElm,
      nodeOps.nextSibling(oldElm)
    )

    // update parent placeholder node element, recursively
    if (isDef(vnode.parent)) {
      let ancestor = vnode.parent
      const patchable = isPatchable(vnode)
      while (ancestor) {
        for (let i = 0; i < cbs.destroy.length; ++i) {
          cbs.destroy[i](ancestor)
        }
        ancestor.elm = vnode.elm
        if (patchable) {
          for (let i = 0; i < cbs.create.length; ++i) {
            cbs.create[i](emptyNode, ancestor)
          }
        }
        // #6513
        // invoke insert hooks that may have been merged by create hooks.
        // e.g. for directives that uses the "inserted" hook.
        const insert = ancestor.data.hook.insert
        if (insert.merged) {
          // start at index 1 to avoid re-invoking component mounted hook
          for (let i = 1; i < insert.fns.length; i++) {
            insert.fns[i]()
          }
        }
      }
    }
  }
}

```

```

        }
    }
} else {
    registerRef(ancestor)
}
ancestor = ancestor.parent
}

// destroy old node
if (isDef(parentElm)) {
    removeVnodes(parentElm, [oldVnode], 0, 0)
} else if (isDef(oldVnode.tag)) {
    invokeDestroyHook(oldVnode)
}
}
}

invokeInsertHook(vnode, insertedVnodeQueue, isInitialPatch)
return vnode.elm
}

```

这里执行 `patch` 的逻辑和首次渲染是不一样的，因为 `oldVnode` 不为空，并且它和 `vnode` 都是 `VNode` 类型，接下来会通过 `sameVNode(oldVnode, vnode)` 判断它们是否是相同的 `VNode` 来决定走不同的更新逻辑：

```

function sameVnode (a, b) {
    return (
        a.key === b.key && (
            (
                a.tag === b.tag &&
                a.isComment === b.isComment &&
                isDef(a.data) === isDef(b.data) &&
                sameInputType(a, b)
            ) || (
                isTrue(a.isAsyncPlaceholder) &&
                a.asyncFactory === b.asyncFactory &&
                isUndef(b.asyncFactory.error)
            )
        )
    )
}

```

`sameVnode` 的逻辑非常简单，如果两个 `vnode` 的 `key` 不相等，则是不同的；否则继续判断对于同步组件，则判断 `isComment`、`data`、`input` 类型等是否相同，对于异步组件，则判断 `asyncFactory` 是否相同。

所以根据新旧 `vnode` 是否为 `sameVnode`，会走到不同的更新逻辑，我们先来说一下不同的情况。

新旧节点不同

如果新旧 `vnode` 不同，那么更新的逻辑非常简单，它本质上是要替换已存在的节点，大致分为 3 步

- 创建新节点

```
const oldElm = oldVnode.elm
const parentElm = nodeOps.parentNode(oldElm)
// create new node
createElm(
  vnode,
  insertedVnodeQueue,
  // extremely rare edge case: do not insert if old element is in a
  // leaving transition. Only happens when combining  transition +
  // keep-alive + HOCs. (#4590)
  oldElm._leaveCb ? null : parentElm,
  nodeOps.nextSibling(oldElm)
)
```

以当前旧节点为参考节点，创建新的节点，并插入到 DOM 中，`createElm` 的逻辑我们之前分析过。

- 更新父的占位符节点

```
// update parent placeholder node element, recursively
if (isDef(vnode.parent)) {
  let ancestor = vnode.parent
  const patchable = isPatchable(vnode)
  while (ancestor) {
    for (let i = 0; i < cbs.destroy.length; ++i) {
      cbs.destroy[i](ancestor)
    }
    ancestor.elm = vnode.elm
    if (patchable) {
      for (let i = 0; i < cbs.create.length; ++i) {
        cbs.create[i](emptyNode, ancestor)
      }
      // #6513
      // invoke insert hooks that may have been merged by create hooks.
      // e.g. for directives that uses the "inserted" hook.
      const insert = ancestor.data.hook.insert
      if (insert.merged) {
        // start at index 1 to avoid re-invoking component mounted hook
        for (let i = 1; i < insert.fns.length; i++) {
          insert.fns[i]()
        }
      }
    } else {
      registerRef(ancestor)
    }
    ancestor = ancestor.parent
  }
}
```

```

    }
}

```

我们只关注主要逻辑即可，找到当前 `vnode` 的父的占位符节点，先执行各个 `module` 的 `destroy` 的钩子函数，如果当前占位符是一个可挂载的节点，则执行 `module` 的 `create` 钩子函数。对于这些钩子函数的作用，在之后的章节会详细介绍。

- 删除旧节点

```

// destroy old node
if (isDef(parentElm)) {
  removeVnodes(parentElm, [oldVnode], 0, 0)
} else if (isDef(oldVnode.tag)) {
  invokeDestroyHook(oldVnode)
}

```

把 `oldVnode` 从当前 DOM 树中删除，如果父节点存在，则执行 `removeVnodes` 方法：

```

function removeVnodes (parentElm, vnodes, startIdx, endIdx) {
  for (; startIdx <= endIdx; ++startIdx) {
    const ch = vnodes[startIdx]
    if (isDef(ch)) {
      if (isDef(ch.tag)) {
        removeAndInvokeRemoveHook(ch)
        invokeDestroyHook(ch)
      } else { // Text node
        removeNode(ch.elm)
      }
    }
  }
}

function removeAndInvokeRemoveHook (vnode, rm) {
  if (isDef(rm) || isDef(vnode.data)) {
    let i
    const listeners = cbs.remove.length + 1
    if (isDef(rm)) {
      // we have a recursively passed down rm callback
      // increase the listeners count
      rm.listeners += listeners
    } else {
      // directly removing
      rm = createRmCb(vnode.elm, listeners)
    }
    // recursively invoke hooks on child component root node
    if (isDef(i = vnode.componentInstance) && isDef(i = i._vnode) && isDef(i.data))
    {
      removeAndInvokeRemoveHook(i, rm)
    }
  }
}

```

```

    for (i = 0; i < cbs.remove.length; ++i) {
      cbs.remove[i](vnode, rm)
    }
    if (isDef(i = vnode.data.hook) && isDef(i = i.remove)) {
      i(vnode, rm)
    } else {
      rm()
    }
  } else {
    removeNode(vnode.elm)
  }
}

function invokeDestroyHook (vnode) {
  let i, j
  const data = vnode.data
  if (isDef(data)) {
    if (isDef(i = data.hook) && isDef(i = i.destroy)) i(vnode)
    for (i = 0; i < cbs.destroy.length; ++i) cbs.destroy[i](vnode)
  }
  if (isDef(i = vnode.children)) {
    for (j = 0; j < vnode.children.length; ++j) {
      invokeDestroyHook(vnode.children[j])
    }
  }
}
}

```

删除节点逻辑很简单，就是遍历待删除的 `vnodes` 做删除，其中 `removeAndInvokeRemoveHook` 的作用是从 DOM 中移除节点并执行 `module` 的 `remove` 钩子函数，并对它的子节点递归调用 `removeAndInvokeRemoveHook` 函数；`invokeDestroyHook` 是执行 `module` 的 `destroy` 钩子函数以及 `vnode` 的 `destroy` 钩子函数，并对它的子 `vnode` 递归调用 `invokeDestroyHook` 函数；`removeNode` 就是调用平台的 DOM API 去把真正的 DOM 节点移除。

在之前介绍组件生命周期的时候提到 `beforeDestroy` & `destroyed` 这两个生命周期钩子函数，它们就是在执行 `invokeDestroyHook` 过程中，执行了 `vnode` 的 `destroy` 钩子函数，它的定义在 `src/core/vdom/create-component.js` 中：

```

const componentVNodeHooks = {
  destroy (vnode: MountedComponentVNode) {
    const { componentInstance } = vnode
    if (!componentInstance._isDestroyed) {
      if (!vnode.data.keepAlive) {
        componentInstance.$destroy()
      } else {
        deactivateChildComponent(componentInstance, true /* direct */)
      }
    }
  }
}

```

当组件并不是 `keepAlive` 的时候，会执行 `componentInstance.$destroy()` 方法，然后就会执行 `beforeDestroy & destroyed` 两个钩子函数。

新旧节点相同

对于新旧节点不同的情况，这种创建新节点 -> 更新占位符节点 -> 删除旧节点的逻辑是很容易理解的。还有一种组件 `vnode` 的更新情况是新旧节点相同，它会调用 `patchVNode` 方法，它的定义在 `src/core/vdom/patch.js` 中：

```
function patchVnode (oldVnode, vnode, insertedVnodeQueue, removeOnly) {
  if (oldVnode === vnode) {
    return
  }

  const elm = vnode.elm = oldVnode.elm

  if (isTrue(oldVnode.isAsyncPlaceholder)) {
    if (isDef(vnode.asyncFactory.resolved)) {
      hydrate(oldVnode.elm, vnode, insertedVnodeQueue)
    } else {
      vnode.isAsyncPlaceholder = true
    }
    return
  }

  // reuse element for static trees.
  // note we only do this if the vnode is cloned -
  // if the new node is not cloned it means the render functions have been
  // reset by the hot-reload-api and we need to do a proper re-render.
  if (isTrue(vnode.isStatic) &&
    isTrue(oldVnode.isStatic) &&
    vnode.key === oldVnode.key &&
    (isTrue(vnode.isCloned) || isTrue(vnode.isOnce)))
  ) {
    vnode.componentInstance = oldVnode.componentInstance
    return
  }

  let i
  const data = vnode.data
  if (isDef(data) && isDef(i = data.hook) && isDef(i = i.prepatch)) {
    i(oldVnode, vnode)
  }

  const oldCh = oldVnode.children
  const ch = vnode.children
  if (isDef(data) && isPatchable(vnode)) {
    for (i = 0; i < cbs.update.length; ++i) cbs.update[i](oldVnode, vnode)
    if (isDef(i = data.hook) && isDef(i = i.update)) i(oldVnode, vnode)
  }
}
```

```

    }
    if (isUndef(vnode.text)) {
      if (isDef(oldCh) && isDef(ch)) {
        if (oldCh !== ch) updateChildren(elm, oldCh, ch, insertedVnodeQueue, removeOn
ly)
      } else if (isDef(ch)) {
        if (isDef(oldVnode.text)) nodeOps.setTextContent(elm, '')
        addVnodes(elm, null, ch, 0, ch.length - 1, insertedVnodeQueue)
      } else if (isDef(oldCh)) {
        removeVnodes(elm, oldCh, 0, oldCh.length - 1)
      } else if (isDef(oldVnode.text)) {
        nodeOps.setTextContent(elm, '')
      }
    } else if (oldVnode.text !== vnode.text) {
      nodeOps.setTextContent(elm, vnode.text)
    }
    if (isDef(data)) {
      if (isDef(i = data.hook) && isDef(i = i.postpatch)) i(oldVnode, vnode)
    }
  }
}

```

`patchVnode` 的作用就是把新的 `vnode` `patch` 到旧的 `vnode` 上，这里我们只关注关键的核心逻辑，我把它拆成四步骤：

- 执行 `prepatch` 钩子函数

```

let i
const data = vnode.data
if (isDef(data) && isDef(i = data.hook) && isDef(i = i.prepatch)) {
  i(oldVnode, vnode)
}

```

当更新的 `vnode` 是一个组件 `vnode` 的时候，会执行 `prepatch` 的方法，它的定义在 `src/core/vdom/create-component.js` 中：

```

const componentVNodeHooks = {
  prepatch (oldVnode: MountedComponentVNode, vnode: MountedComponentVNode) {
    const options = vnode.componentInstance.options
    const child = vnode.componentInstance = oldVnode.componentInstance
    updateChildComponent(
      child,
      options.propsData, // updated props
      options.listeners, // updated listeners
      vnode, // new parent vnode
      options.children // new children
    )
  }
}

```

`prepatch` 方法就是拿到新的 `vnode` 的组件配置以及组件实例，去执行 `updateChildComponent` 方法，它的定义在 `src/core/instance/lifecycle.js` 中：

```

export function updateChildComponent (
  vm: Component,
  propsData: ?Object,
  listeners: ?Object,
  parentVnode: MountedComponentVNode,
  renderChildren: ?Array<VNode>
) {
  if (process.env.NODE_ENV !== 'production') {
    isUpdatingChildComponent = true
  }

  // determine whether component has slot children
  // we need to do this before overwriting $options._renderChildren
  const hasChildren = !!(
    renderChildren || // has new static slots
    vm.$options._renderChildren || // has old static slots
    parentVnode.data.scopedSlots || // has new scoped slots
    vm.$scopedSlots !== emptyObject // has old scoped slots
  )

  vm.$options._parentVnode = parentVnode
  vm.$vnode = parentVnode // update vm's placeholder node without re-render

  if (vm._vnode) { // update child tree's parent
    vm._vnode.parent = parentVnode
  }
  vm.$options._renderChildren = renderChildren

  // update $attrs and $listeners hash
  // these are also reactive so they may trigger child update if the child
  // used them during render
  vm.$attrs = parentVnode.data.attrs || emptyObject
  vm.$listeners = listeners || emptyObject

  // update props
  if (propsData && vm.$options.props) {
    toggleObserving(false)
    const props = vm._props
    const propKeys = vm.$options._propKeys || []
    for (let i = 0; i < propKeys.length; i++) {
      const key = propKeys[i]
      const propOptions: any = vm.$options.props // wtf flow?
      props[key] = validateProp(key, propOptions, propsData, vm)
    }
    toggleObserving(true)
    // keep a copy of raw propsData
    vm.$options.propsData = propsData
  }
}

```

```

    }

    // update listeners
    listeners = listeners || emptyObject
    const oldListeners = vm.$options._parentListeners
    vm.$options._parentListeners = listeners
    updateComponentListeners(vm, listeners, oldListeners)

    // resolve slots + force update if has children
    if (hasChildren) {
        vm.$slots = resolveSlots(renderChildren, parentVnode.context)
        vm.$forceUpdate()
    }

    if (process.env.NODE_ENV !== 'production') {
        isUpdatingChildComponent = false
    }
}

```

`updateChildComponent` 的逻辑也非常简单，由于更新了 `vnode`，那么 `vnode` 对应的实例 `vm` 的一系列属性也会发生变化，包括占位符 `vm.$vnode` 的更新、`slot` 的更新、`listeners` 的更新，`props` 的更新等等。

- 执行 `update` 钩子函数

```

if (isDef(data) && isPatchable(vnode)) {
    for (i = 0; i < cbs.update.length; ++i) cbs.update[i](oldVnode, vnode)
    if (isDef(i = data.hook) && isDef(i = i.update)) i(oldVnode, vnode)
}

```

回到 `patchVNode` 函数，在执行完新的 `vnode` 的 `prepatch` 钩子函数，会执行所有 `module` 的 `update` 钩子函数以及用户自定义 `update` 钩子函数，对于 `module` 的钩子函数，之后我们会有具体的章节针对一些具体的 case 分析。

- 完成 `patch` 过程

```

const oldCh = oldVnode.children
const ch = vnode.children
if (isDef(data) && isPatchable(vnode)) {
    for (i = 0; i < cbs.update.length; ++i) cbs.update[i](oldVnode, vnode)
    if (isDef(i = data.hook) && isDef(i = i.update)) i(oldVnode, vnode)
}
if (isUndef(vnode.text)) {
    if (isDef(oldCh) && isDef(ch)) {
        if (oldCh !== ch) updateChildren(elm, oldCh, ch, insertedVnodeQueue, removeOnly)
    }
} else if (isDef(ch)) {
    if (isDef(oldVnode.text)) nodeOps.setTextContent(elm, '')
    addVnodes(elm, null, ch, 0, ch.length - 1, insertedVnodeQueue)
}

```

```

} else if (isDef(oldCh)) {
  removeVnodes(elm, oldCh, 0, oldCh.length - 1)
} else if (isDef(oldVnode.text)) {
  nodeOps.setTextContent(elm, '')
}
} else if (oldVnode.text !== vnode.text) {
  nodeOps.setTextContent(elm, vnode.text)
}

```

如果 `vnode` 是个文本节点且新旧文本不相同，则直接替换文本内容。如果不是文本节点，则判断它们的子节点，并分了几种情况处理：

1. `oldCh` 与 `ch` 都存在且不相时，使用 `updateChildren` 函数来更新子节点，这个后面重点讲。
2. 如果只有 `ch` 存在，表示旧节点不需要了。如果旧的节点是文本节点则先将节点的文本清除，然后通过 `addVnodes` 将 `ch` 批量插入到新节点 `elm` 下。
3. 如果只有 `oldCh` 存在，表示更新的是空节点，则需要将旧的节点通过 `removeVnodes` 全部清除。
4. 当只有旧节点是文本节点的时候，则清除其节点文本内容。

- 执行 `postpatch` 钩子函数

```

if (isDef(data)) {
  if (isDef(i = data.hook) && isDef(i = i.postpatch)) i(oldVnode, vnode)
}

```

再执行完 `patch` 过程后，会执行 `postpatch` 钩子函数，它是组件自定义的钩子函数，有则执行。

那么在整个 `patchVnode` 过程中，最复杂的就是 `updateChildren` 方法了，下面我们来单独介绍它。

updateChildren

```

function updateChildren (parentElm, oldCh, newCh, insertedVnodeQueue, removeOnly) {
  let oldStartIdx = 0
  let newStartIdx = 0
  let oldEndIdx = oldCh.length - 1
  let oldStartVnode = oldCh[0]
  let oldEndVnode = oldCh[oldEndIdx]
  let newEndIdx = newCh.length - 1
  let newStartVnode = newCh[0]
  let newEndVnode = newCh[newEndIdx]
  let oldKeyToIdx, idxInOld, vnodeToMove, refElm

  // removeOnly is a special flag used only by <transition-group>
  // to ensure removed elements stay in correct relative positions
  // during leaving transitions
}

```

```

const canMove = !removeOnly

if (process.env.NODE_ENV !== 'production') {
  checkDuplicateKeys(newCh)
}

while (oldStartIdx <= oldEndIdx && newStartIdx <= newEndIdx) {
  if (isUndef(oldStartVnode)) {
    oldStartVnode = oldCh[++oldStartIdx] // Vnode has been moved left
  } else if (isUndef(oldEndVnode)) {
    oldEndVnode = oldCh[--oldEndIdx]
  } else if (sameVnode(oldStartVnode, newStartVnode)) {
    patchVnode(oldStartVnode, newStartVnode, insertedVnodeQueue)
    oldStartVnode = oldCh[++oldStartIdx]
    newStartVnode = newCh[++newStartIdx]
  } else if (sameVnode(oldEndVnode, newEndVnode)) {
    patchVnode(oldEndVnode, newEndVnode, insertedVnodeQueue)
    oldEndVnode = oldCh[--oldEndIdx]
    newEndVnode = newCh[--newEndIdx]
  } else if (sameVnode(oldStartVnode, newEndVnode)) { // Vnode moved right
    patchVnode(oldStartVnode, newEndVnode, insertedVnodeQueue)
    canMove && nodeOps.insertBefore(parentElm, oldStartVnode.elm, nodeOps.nextSibling(oldEndVnode.elm))
    oldStartVnode = oldCh[++oldStartIdx]
    newEndVnode = newCh[--newEndIdx]
  } else if (sameVnode(oldEndVnode, newStartVnode)) { // Vnode moved left
    patchVnode(oldEndVnode, newStartVnode, insertedVnodeQueue)
    canMove && nodeOps.insertBefore(parentElm, oldEndVnode.elm, oldStartVnode.elm)
  }
  oldEndVnode = oldCh[--oldEndIdx]
  newStartVnode = newCh[++newStartIdx]
} else {
  if (isUndef(oldKeyToIdx)) oldKeyToIdx = createKeyToOldIdx(oldCh, oldStartIdx, oldEndIdx)
  idxInOld = isDef(newStartVnode.key)
    ? oldKeyToIdx[newStartVnode.key]
    : findIdxInOld(newStartVnode, oldCh, oldStartIdx, oldEndIdx)
  if (isUndef(idxInOld)) { // New element
    createElm(newStartVnode, insertedVnodeQueue, parentElm, oldStartVnode.elm,
false, newCh, newStartIdx)
  } else {
    vnodeToMove = oldCh[idxInOld]
    if (sameVnode(vnodeToMove, newStartVnode)) {
      patchVnode(vnodeToMove, newStartVnode, insertedVnodeQueue)
      oldCh[idxInOld] = undefined
      canMove && nodeOps.insertBefore(parentElm, vnodeToMove.elm, oldStartVnode.elm)
    } else {
      // same key but different element. treat as new element
      createElm(newStartVnode, insertedVnodeQueue, parentElm, oldStartVnode.elm
, false, newCh, newStartIdx)
    }
  }
}

```

```

        }
    }
    newStartVnode = newCh[++newStartIdx]
}
}
if (oldStartIdx > oldEndIdx) {
    refElm = isUndef(newCh[newEndIdx + 1]) ? null : newCh[newEndIdx + 1].elm
    addVnodes(parentElm, refElm, newCh, newStartIdx, newEndIdx, insertedVnodeQueue)
} else if (newStartIdx > newEndIdx) {
    removeVnodes(parentElm, oldCh, oldStartIdx, oldEndIdx)
}
}

```

`updateChildren` 的逻辑比较复杂，直接读源码比较晦涩，我们可以通过一个具体的示例来分析它。

```

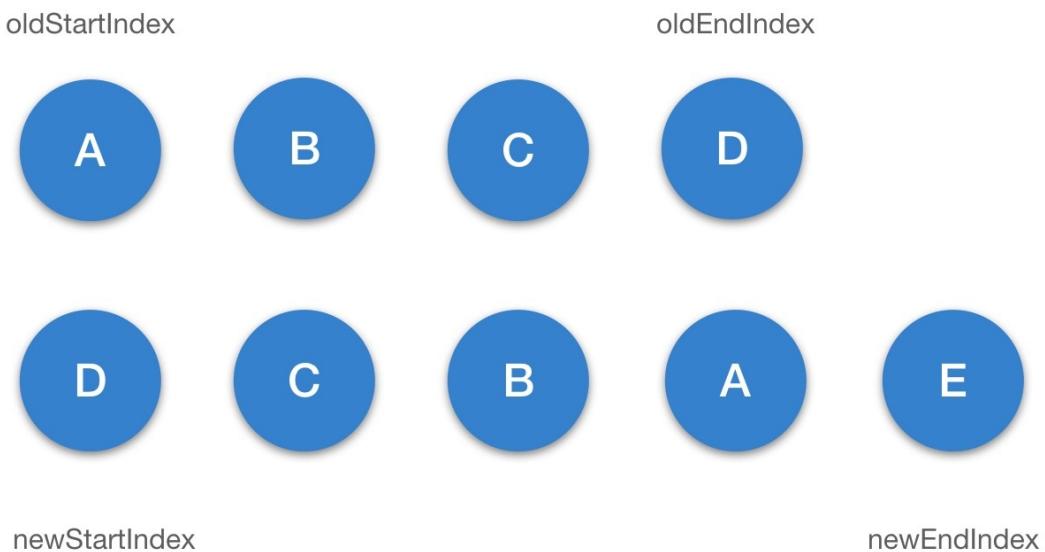
<template>
<div id="app">
<div>
<ul>
    <li v-for="item in items" :key="item.id">{{ item.val }}</li>
</ul>
</div>
<button @click="change">change</button>
</div>
</template>

<script>
export default {
  name: 'App',
  data() {
    return {
      items: [
        {id: 0, val: 'A'},
        {id: 1, val: 'B'},
        {id: 2, val: 'C'},
        {id: 3, val: 'D'}
      ]
    }
  },
  methods: {
    change() {
      this.items.reverse().push({id: 4, val: 'E'})
    }
  }
}
</script>

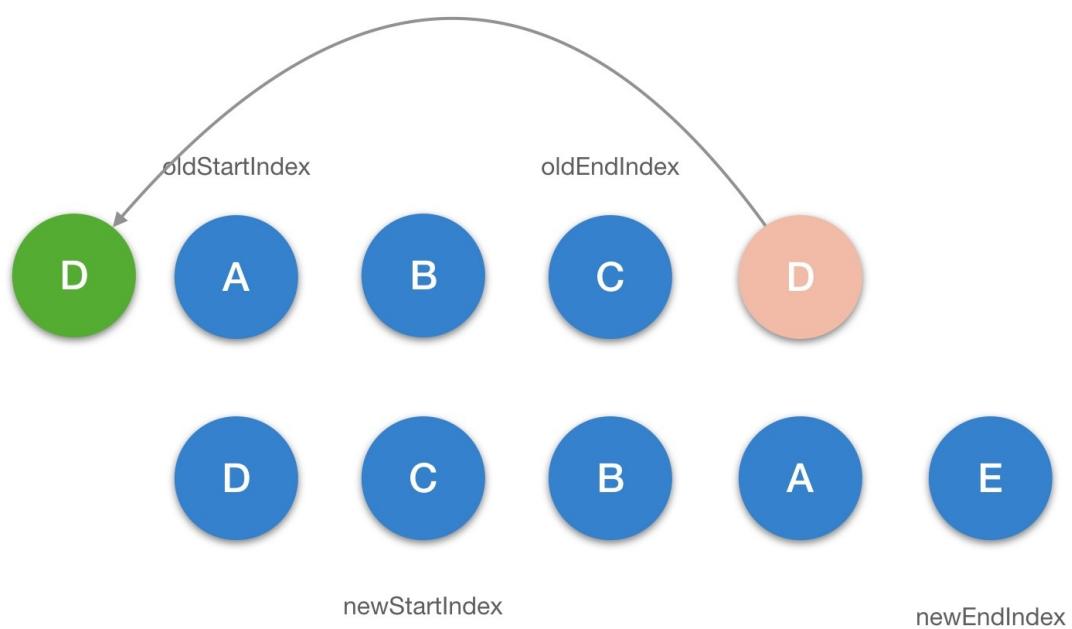
```

当我们点击 `change` 按钮去改变数据，最终会执行到 `updateChildren` 去更新 `li` 部分的列表数据，我们通过图的方式来描述一下它的更新过程：

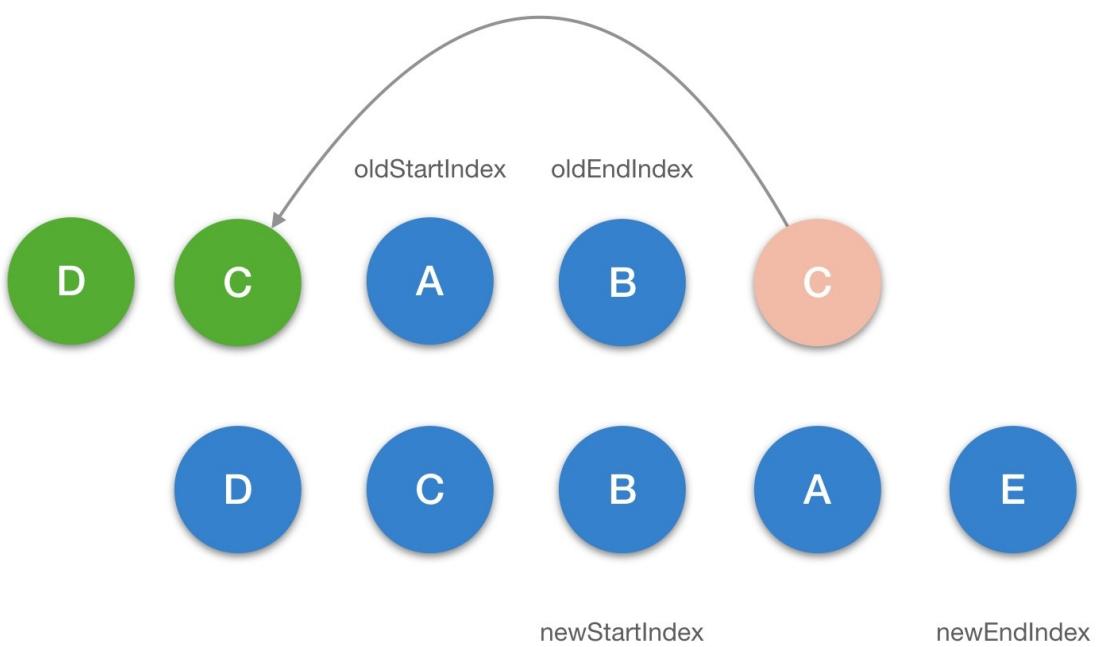
第一步：



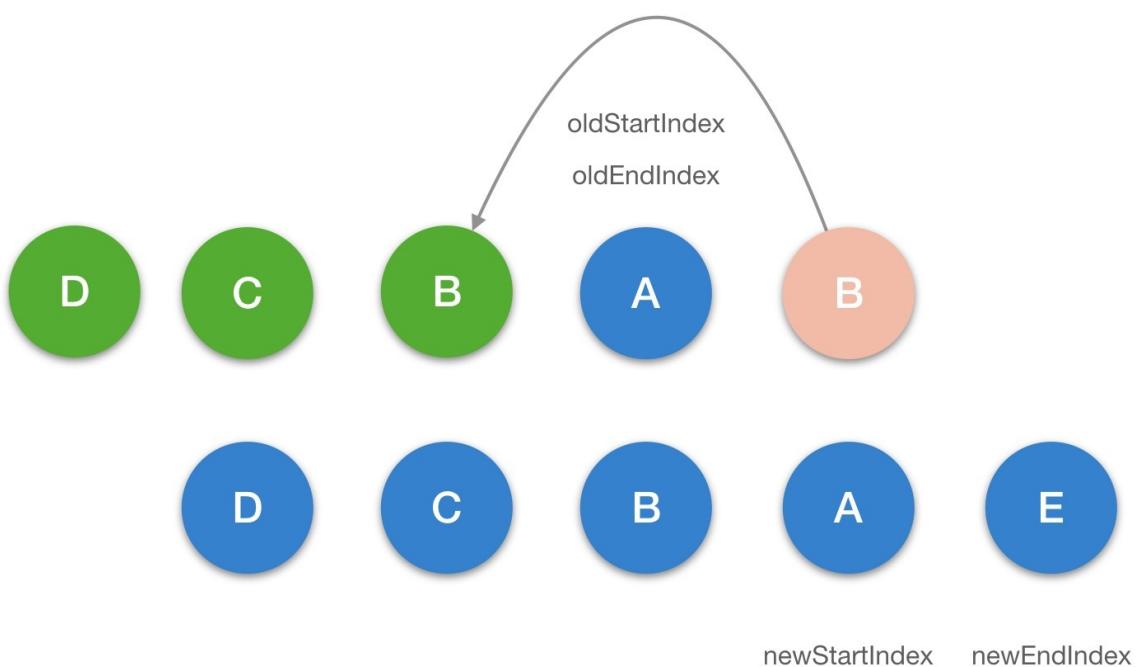
第二步：



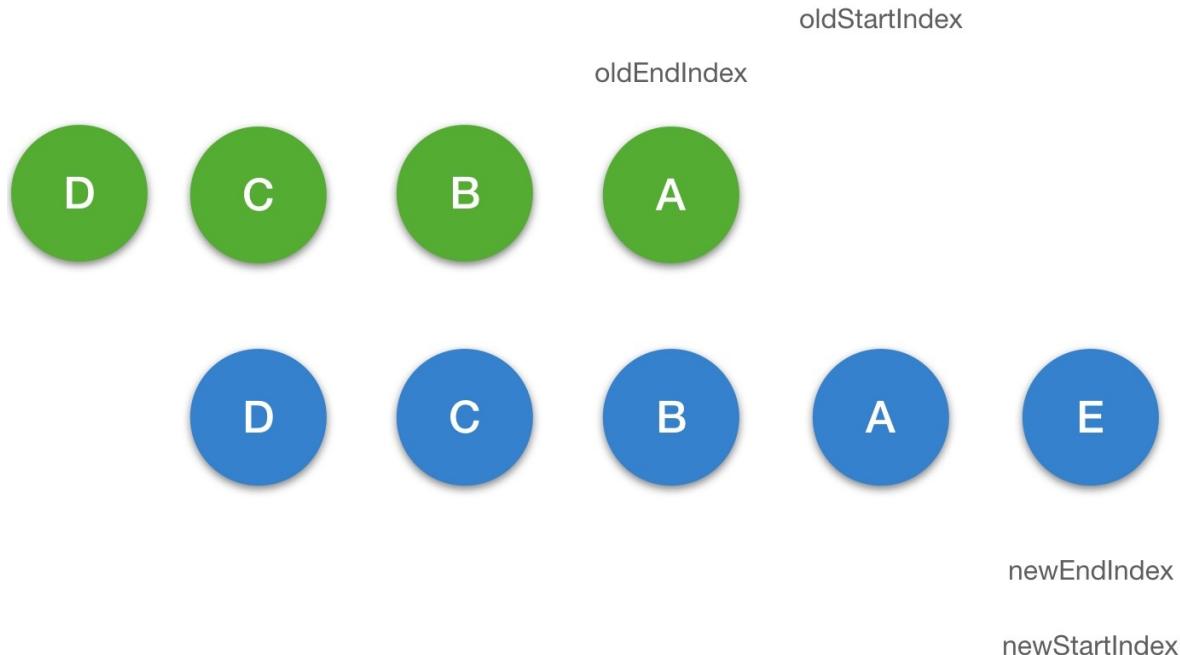
第三步：



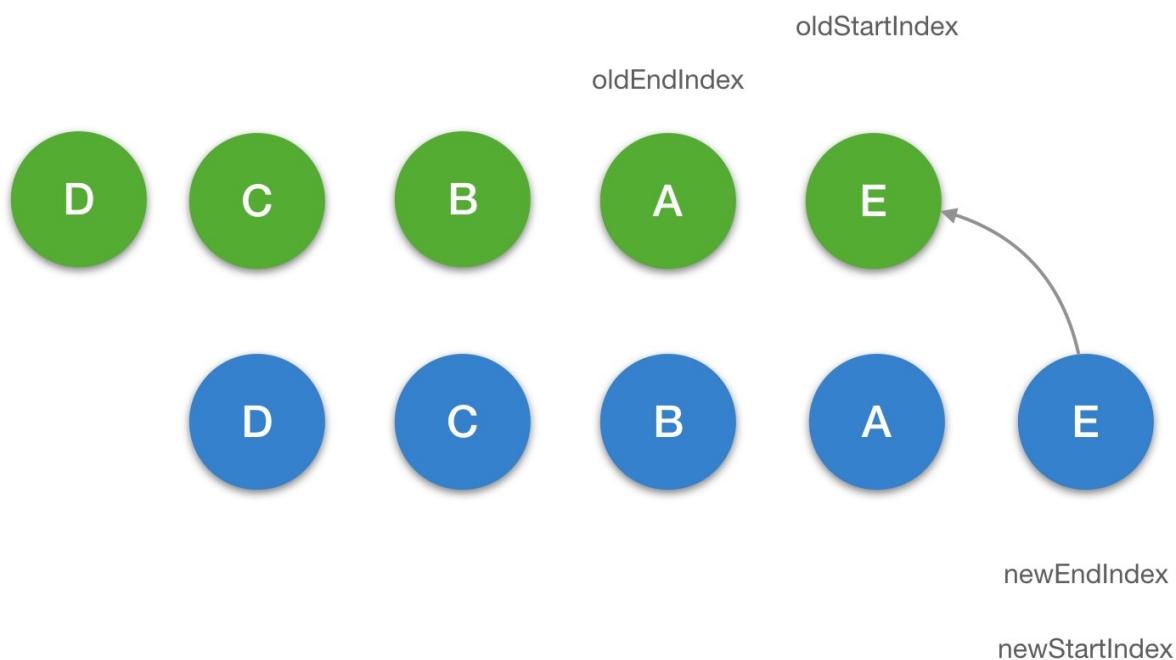
第四步：



第五步：



第六步：

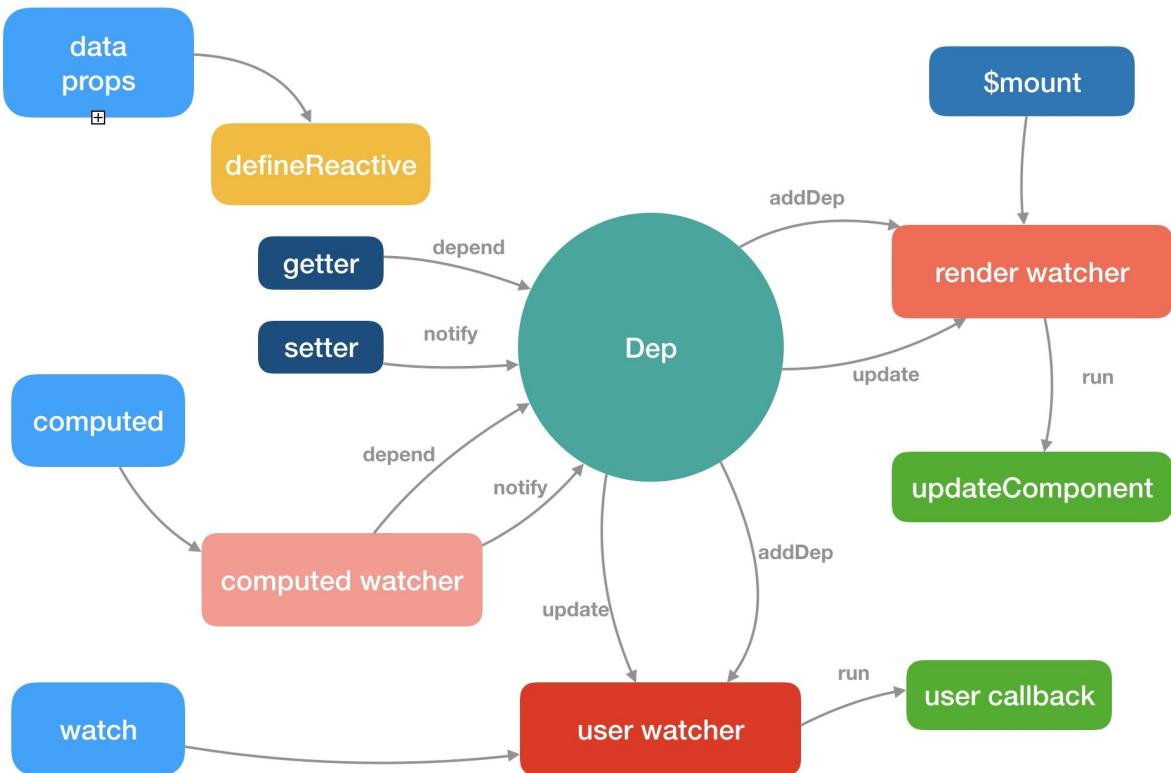


总结

组件更新的过程核心就是新旧 vnode diff，对新旧节点相同以及不同的情况分别做不同的处理。新旧节点不同的更新流程是创建新节点->更新父占位符节点->删除旧节点；而新旧节点相同的更新流程是去获取它们的 children，根据不同情况做不同的更新逻辑。最复杂的情况是新旧节点相同且它们都存在子节

点，那么会执行 `updateChildren` 逻辑，这块儿可以借助画图的方式配合理解。

原理图



编译

之前我们分析过模板到真实 DOM 渲染的过程，中间有一个环节是把模板编译成 `render` 函数，这个过程我们把它称作编译。

虽然我们可以直接为组件编写 `render` 函数，但是编写 `template` 模板更加直观，也更符合我们的开发习惯。

Vue.js 提供了 2 个版本，一个是 Runtime + Compiler 的，一个是 Runtime only 的，前者是包含编译代码的，可以把编译过程放在运行时做，后者是不包含编译代码的，需要借助 webpack 的 `vue-loader` 事先把模板编译成 `render` 函数。

这一章我们就来分析编译的过程，对编译过程的了解会让我们对 Vue 的指令、内置组件等有更好的理解。不过由于编译的过程是一个相对复杂的过程，我们只要求理解整体的流程、输入和输出即可，对于细节我们不必抠太细。有些细节比如对于 `slot` 的处理我们可以在之后去分析插槽实现的时候再详 细分析。

编译入口

当我们使用 Runtime + Compiler 的 Vue.js，它的入口是 `src/platforms/web/entry-runtime-with-compiler.js`，看一下它对 `$mount` 函数的定义：

```

const mount = Vue.prototype.$mount
Vue.prototype.$mount = function (
  el?: string | Element,
  hydrating?: boolean
): Component {
  el = el && query(el)

  /* istanbul ignore if */
  if (el === document.body || el === document.documentElement) {
    process.env.NODE_ENV !== 'production' && warn(
      `Do not mount Vue to <html> or <body> - mount to normal elements instead.`
    )
    return this
  }

  const options = this.$options
  // resolve template/el and convert to render function
  if (!options.render) {
    let template = options.template
    if (template) {
      if (typeof template === 'string') {
        if (template.charAt(0) === '#') {
          template = idToTemplate(template)
          /* istanbul ignore if */
          if (process.env.NODE_ENV !== 'production' && !template) {
            warn(
              `Template element not found or is empty: ${options.template}`,
              this
            )
          }
        }
      } else if (template.nodeType) {
        template = template.innerHTML
      } else {
        if (process.env.NODE_ENV !== 'production') {
          warn('invalid template option:' + template, this)
        }
        return this
      }
    } else if (el) {
      template = getOuterHTML(el)
    }
    if (template) {
  
```

```

/* istanbul ignore if */
if (process.env.NODE_ENV !== 'production' && config.performance && mark) {
  mark('compile')
}

const { render, staticRenderFns } = compileToFunctions(template, {
  shouldDecodeNewlines,
  shouldDecodeNewlinesForHref,
  delimiters: options.delimiters,
  comments: options.comments
}, this)
options.render = render
options.staticRenderFns = staticRenderFns

/* istanbul ignore if */
if (process.env.NODE_ENV !== 'production' && config.performance && mark) {
  mark('compile end')
  measure(`vue ${this._name} compile`, 'compile', 'compile end')
}
}

return mount.call(this, el, hydrating)
}

```

这段函数逻辑之前分析过，关于编译的入口就是在这里：

```

const { render, staticRenderFns } = compileToFunctions(template, {
  shouldDecodeNewlines,
  shouldDecodeNewlinesForHref,
  delimiters: options.delimiters,
  comments: options.comments
}, this)
options.render = render
options.staticRenderFns = staticRenderFns

```

`compileToFunctions` 方法就是把模板 `template` 编译生成 `render` 以及 `staticRenderFns`，它的定义在 `src/platforms/web/compiler/index.js` 中：

```

import { baseOptions } from './options'
import { createCompiler } from 'compiler/index'

const { compile, compileToFunctions } = createCompiler(baseOptions)

export { compile, compileToFunctions }

```

可以看到 `compileToFunctions` 方法实际上是 `createCompiler` 方法的返回值，该方法接收一个编译配置参数，接下来我们来看一下 `createCompiler` 方法的定义，在 `src/compiler/index.js` 中：

```
// `createCompilerCreator` allows creating compilers that use alternative
// parser/optimizer/codegen, e.g the SSR optimizing compiler.
// Here we just export a default compiler using the default parts.
export const createCompiler = createCompilerCreator(function baseCompile (
  template: string,
  options: CompilerOptions
): CompiledResult {
  const ast = parse(template.trim(), options)
  if (options.optimize !== false) {
    optimize(ast, options)
  }
  const code = generate(ast, options)
  return {
    ast,
    render: code.render,
    staticRenderFns: code.staticRenderFns
  }
})
```

`createCompiler` 方法实际上是通过调用 `createCompilerCreator` 方法返回的，该方法传入的参数是一个函数，真正的编译过程都在这个 `baseCompile` 函数里执行，那么 `createCompilerCreator` 又是什么呢，它的定义在 `src/compiler/create-compiler.js` 中：

```
export function createCompilerCreator (baseCompile: Function): Function {
  return function createCompiler (baseOptions: CompilerOptions) {
    function compile (
      template: string,
      options?: CompilerOptions
    ): CompiledResult {
      const finalOptions = Object.create(baseOptions)
      const errors = []
      const tips = []
      finalOptions.warn = (msg, tip) => {
        (tip ? tips : errors).push(msg)
      }

      if (options) {
        // merge custom modules
        if (options.modules) {
          finalOptions.modules =
            (baseOptions.modules || []).concat(options.modules)
        }
        // merge custom directives
        if (options.directives) {
          finalOptions.directives = extend(
            Object.create(baseOptions.directives || null),
            options.directives
          )
        }
      }
    }
  }
}
```

```

    // copy other options
    for (const key in options) {
      if (key !== 'modules' && key !== 'directives') {
        finalOptions[key] = options[key]
      }
    }
  }

  const compiled = baseCompile(template, finalOptions)
  if (process.env.NODE_ENV !== 'production') {
    errors.push.apply(errors, detectErrors(compiled.ast))
  }
  compiled.errors = errors
  compiled.tips = tips
  return compiled
}

return {
  compile,
  compileToFunctions: createCompileToFunctionFn(compile)
}
}
}

```

可以看到该方法返回了一个 `createCompiler` 的函数，它接收一个 `baseOptions` 的参数，返回的是一个对象，包括 `compile` 方法属性和 `compileToFunctions` 属性，这个 `compileToFunctions` 对应的就是 `$mount` 函数调用的 `compileToFunctions` 方法，它是调用

`createCompileToFunctionFn` 方法的返回值，我们接下来看一下 `createCompileToFunctionFn` 方法，它的定义在 `src/compiler/to-function/js` 中：

```

export function createCompileToFunctionFn (compile: Function): Function {
  const cache = Object.create(null)

  return function compileToFunctions (
    template: string,
    options?: CompilerOptions,
    vm?: Component
  ): CompiledFunctionResult {
    options = extend({}, options)
    const warn = options.warn || baseWarn
    delete options.warn

    /* istanbul ignore if */
    if (process.env.NODE_ENV !== 'production') {
      // detect possible CSP restriction
      try {
        new Function('return 1')
      } catch (e) {
        if (e.toString().match(/unsafe-eval|CSP/)) {

```

```

    warn(
      'It seems you are using the standalone build of Vue.js in an ' +
      'environment with Content Security Policy that prohibits unsafe-eval. ' +
      '+
      'The template compiler cannot work in this environment. Consider ' +
      'relaxing the policy to allow unsafe-eval or pre-compiling your ' +
      'templates into render functions.'
    )
  }
}

// check cache
const key = options.delimiters
  ? String(options.delimiters) + template
  : template
if (cache[key]) {
  return cache[key]
}

// compile
const compiled = compile(template, options)

// check compilation errors/tips
if (process.env.NODE_ENV !== 'production') {
  if (compiled.errors && compiled.errors.length) {
    warn(
      `Error compiling template:\n\n${template}\n\n` +
      compiled.errors.map(e => `- ${e}`).join('\n') + '\n',
      vm
    )
  }
  if (compiled.tips && compiled.tips.length) {
    compiled.tips.forEach(msg => tip(msg, vm))
  }
}

// turn code into functions
const res = {}
const fnGenErrors = []
res.render = createFunction(compiled.render, fnGenErrors)
res.staticRenderFns = compiled.staticRenderFns.map(code => {
  return createFunction(code, fnGenErrors)
})

// check function generation errors.
// this should only happen if there is a bug in the compiler itself.
// mostly for codegen development use
/* istanbul ignore if */
if (process.env.NODE_ENV !== 'production') {
  if ((!compiled.errors || !compiled.errors.length) && fnGenErrors.length) {
}

```

```

        warn(
            `Failed to generate render function:\n\n` +
            fnGenErrors.map(({ err, code }) => `${err.toString()} in\n\n${code}\n`).join(
                '\n',
                vm
            )
        )
    }

    return (cache[key] = res)
}
}

```

至此我们总算找到了 `compileToFunctions` 的最终定义，它接收 3 个参数、编译模板 `template`，编译配置 `options` 和 Vue 实例 `vm`。核心的编译过程就一行代码：

```
const compiled = compile(template, options)
```

`compile` 函数在执行 `createCompileToFunctionFn` 的时候作为参数传入，它是 `createCompiler` 函数中定义的 `compile` 函数，如下：

```

function compile (
    template: string,
    options?: CompilerOptions
): CompiledResult {
    const finalOptions = Object.create(baseOptions)
    const errors = []
    const tips = []
    finalOptions.warn = (msg, tip) => {
        (tip ? tips : errors).push(msg)
    }

    if (options) {
        // merge custom modules
        if (options.modules) {
            finalOptions.modules =
                (baseOptions.modules || []).concat(options.modules)
        }
        // merge custom directives
        if (options.directives) {
            finalOptions.directives = extend(
                Object.create(baseOptions.directives || null),
                options.directives
            )
        }
        // copy other options
        for (const key in options) {
            if (key !== 'modules' && key !== 'directives') {
                finalOptions[key] = options[key]
            }
        }
    }
}

```

```

        }
    }
}

const compiled = baseCompile(template, finalOptions)
if (process.env.NODE_ENV !== 'production') {
    errors.push.apply(errors, detectErrors(compiled.ast))
}
compiled.errors = errors
compiled.tips = tips
return compiled
}

```

`compile` 函数执行的逻辑是先处理配置参数，真正执行编译过程就一行代码：

```
const compiled = baseCompile(template, finalOptions)
```

`baseCompile` 在执行 `createCompilerCreator` 方法时作为参数传入，如下：

```

export const createCompiler = createCompilerCreator(function baseCompile (
    template: string,
    options: CompilerOptions
): CompiledResult {
    const ast = parse(template.trim(), options)
    optimize(ast, options)
    const code = generate(ast, options)
    return {
        ast,
        render: code.render,
        staticRenderFns: code.staticRenderFns
    }
})

```

所以编译的入口我们终于找到了，它主要就是执行了如下几个逻辑：

- 解析模板字符串生成 AST

```
const ast = parse(template.trim(), options)
```

- 优化语法树

```
optimize(ast, options)
```

- 生成代码

```
const code = generate(ast, options)
```

那么接下来的章节我会带大家去逐步分析这几个过程。

总结

编译入口逻辑之所以这么绕，是因为 Vue.js 在不同的平台下都会有编译的过程，因此编译过程中的依赖的配置 `baseOptions` 会有所不同。而编译过程会多次执行，但这同一个平台下每一次的编译过程配置又是相同的，为了不让这些配置在每次编译过程都通过参数传入，Vue.js 利用了函数柯里化的技巧很好的实现了 `baseOptions` 的参数保留。同样，Vue.js 也是利用函数柯里化技巧把基础的编译过程函数抽出来，通过 `createCompilerCreator(baseCompile)` 的方式把真正编译的过程和其它逻辑如对编译配置处理、缓存处理等剥离开，这样的设计还是非常巧妙的。

parse

编译过程首先就是对模板做解析，生成 AST，它是一种抽象语法树，是对源代码的抽象语法结构的树状表现形式。在很多编译技术中，如 babel 编译 ES6 的代码都会先生成 AST。

这个过程是比较复杂的，它会用到大量正则表达式对字符串解析，如果对正则不是很了解，建议先去补习正则表达式的知识。为了直观地演示 `parse` 的过程，我们先来看一个例子：

```
<ul :class="bindCls" class="list" v-if="isShow">
  <li v-for="(item, index) in data" @click="clickItem(index)">{{item}}:{{index}}</li>
</ul>
```

经过 `parse` 过程后，生成的 AST 如下：

```
ast = {
  'type': 1,
  'tag': 'ul',
  'attrsList': [],
  'attrsMap': {
    ':class': 'bindCls',
    'class': 'list',
    'v-if': 'isShow'
  },
  'if': 'isShow',
  'ifConditions': [
    {
      'exp': 'isShow',
      'block': // ul ast element
    }
  ],
  'parent': undefined,
  'plain': false,
  'staticClass': 'list',
  'classBinding': 'bindCls',
  'children': [
    {
      'type': 1,
      'tag': 'li',
      'attrsList': [
        {
          'name': '@click',
          'value': 'clickItem(index)'
        }
      ],
      'attrsMap': {
        '@click': 'clickItem(index)',
        'v-for': '(item, index) in data'
      },
      'parent': // li ast element
      'plain': false,
      'events': {
        ...
      }
    }
  ]
}
```

```

    'click': {
      'value': 'clickItem(index)'
    },
    'hasBindings': true,
    'for': 'data',
    'alias': 'item',
    'iterator1': 'index',
    'children': [
      {
        'type': 2,
        'expression': '_s(item)+":"+_s(index)'
        'text': '{{item}}:{{index}}',
        'tokens': [
          {'@binding': 'item'},
          ':',
          {'@binding': 'index'}
        ]
      }
    ]
  }
}

```

可以看到，生成的 AST 是一个树状结构，每一个节点都是一个 `ast element`，除了它自身的一些属性，还维护了它的父子关系，如 `parent` 指向它的父节点，`children` 指向它的所有子节点。先对 AST 有一些直观的印象，那么接下来我们来分析一下这个 AST 是如何得到的。

整体流程

首先来看一下 `parse` 的定义，在 `src/compiler/parser/index.js` 中：

```

export function parse (
  template: string,
  options: CompilerOptions
): ASTEElement | void {
  getFnsAndConfigFromOptions(options)

  parseHTML(template, {
    // options ...
    start (tag, attrs, unary) {
      let element = createASTElement(tag, attrs)
      processElement(element)
      treeManagement()
    },
    end () {
      treeManagement()
      closeElement()
    },
    chars (text: string) {

```

```

        handleText()
        createChildrenASTOfText()
    },
    comment (text: string) {
        createChildrenASTOfComment()
    }
})
return astRootElement
}

```

`parse` 函数的代码很长，贴一遍对同学的理解没有好处，我先把它拆成伪代码的形式，方便同学们对整体流程先有一个大致的了解。接下来我们就来分解分析每段伪代码的作用。

从 `options` 中获取方法和配置

对应伪代码：

```
getFnsAndConfigFromOptions(options)
```

`parse` 函数的输入是 `template` 和 `options`，输出是 AST 的根节点。`template` 就是我们的模板字符串，而 `options` 实际上是和平台相关的一些配置，它的定义在 `src/platforms/web/compiler/options` 中：

```

import {
  isPreTag,
  mustUseProp,
  isReservedTag,
  getTagNameSpace
} from '../util/index'

import modules from './modules/index'
import directives from './directives/index'
import { genStaticKeys } from 'shared/util'
import { isUnaryTag, canBeLeftOpenTag } from './util'

export const baseOptions: CompilerOptions = {
  expectHTML: true,
  modules,
  directives,
  isPreTag,
  isUnaryTag,
  mustUseProp,
  canBeLeftOpenTag,
  isReservedTag,
  getTagNameSpace,
  staticKeys: genStaticKeys(modules)
}

```

这些属性和方法之所以放到 `platforms` 目录下是因为它们在不同的平台（web 和 weex）的实现是不同的。

我们用伪代码 `getFnsAndConfigFromOptions` 表示了这一过程，它的实际代码如下：

```
warn = options.warn || baseWarn

platformIsPreTag = options.isPreTag || no
platformMustUseProp = options.mustUseProp || no
platformGetTagNameSpace = options.getTagNamespace || no

transforms = pluckModuleFunction(options.modules, 'transformNode')
preTransforms = pluckModuleFunction(options.modules, 'preTransformNode')
postTransforms = pluckModuleFunction(options.modules, 'postTransformNode')

delimiters = options.delimiters
```

这些方法和配置都是后续解析时候需要的，可以不用去管它们的具体作用，我们先往后看。

解析 HTML 模板

对应伪代码：

```
parseHTML(template, options)
```

对于 `template` 模板的解析主要是通过 `parseHTML` 函数，它的定义在 `src/compiler/parser/html-parser` 中：

```
export function parseHTML (html, options) {
  let lastTag
  while (html) {
    if (!lastTag || !isPlainTextElement(lastTag)){
      let textEnd = html.indexOf('<')
      if (textEnd === 0) {
        if(matchComment) {
          advance(commentLength)
          continue
        }
        if(matchDoctype) {
          advance(doctypeLength)
          continue
        }
        if(matchEndTag) {
          advance(endTagLength)
          parseEndTag()
          continue
        }
        if(matchStartTag) {
```

```

        parseStartTag()
        handleStartTag()
        continue
    }
}
handleText()
advance(textLength)
} else {
    handlePlainTextElement()
    parseEndTag()
}
}
}
}

```

由于 `parseHTML` 的逻辑也非常复杂，因此我也用了伪代码的方式表达，整体来说它的逻辑就是循环解析 `template`，用正则做各种匹配，对于不同情况分别进行不同的处理，直到整个 `template` 被解析完毕。在匹配的过程中会利用 `advance` 函数不断前进整个模板字符串，直到字符串末尾。

```

function advance (n) {
    index += n
    html = html.substring(n)
}

```

为了更加直观地说明 `advance` 的作用，可以通过一副图表示：



调用 `advance` 函数：

```
advance(4)
```

得到结果：



匹配的过程中主要利用了正则表达式，如下：

```
const attribute = /\^\\s*([\\^\\s\"'<>\\/=]+)(?:\\s*(=)\\s*(?:\"([\\^"]*)\"+|'([\\^']*)'+|([\\^\\s\"'=\\>`]+)))/
const ncname = '[a-zA-Z_][\\w\\-\\.]*'
const qnameCapture = `((?:${ncname}\\:\\:)?${ncname})`
const startTagOpen = new RegExp(`^<${qnameCapture}`)
const startTagClose = /\^\\s*(\\/?)>/
const endTag = new RegExp(`^<\\/$${qnameCapture}[^>]*>`)
const doctype = /^<!DOCTYPE [^>]+>/i
const comment = /^<!--/
const conditionalComment = /^<![/
```

通过这些正则表达式，我们可以匹配注释节点、文档类型节点、开始闭合标签等。

- 注释节点、文档类型节点

对于注释节点和文档类型节点的匹配，如果匹配到我们仅仅做的是做前进即可。

```
if (comment.test(html)) {
  const commentEnd = html.indexOf('---')
  if (commentEnd >= 0) {
    if (options.shouldKeepComment) {
      options.comment(html.substring(4, commentEnd))
    }
    advance(commentEnd + 3)
    continue
  }
}

if (conditionalComment.test(html)) {
  const conditionalEnd = html.indexOf(']>')
  if (conditionalEnd >= 0) {
    advance(conditionalEnd + 2)
    continue
  }
}

const doctypeMatch = html.match(doctype)
if (doctypeMatch) {
  advance(doctypeMatch[0].length)
  continue
}
```

对于注释和条件注释节点，前进至它们的末尾位置；对于文档类型节点，则前进它自身长度的距离。

- 开始标签

```

const startTagMatch = parseStartTag()
if (startTagMatch) {
  handleStartTag(startTagMatch)
  if (shouldIgnoreFirstNewline(lastTag, html)) {
    advance(1)
  }
  continue
}

```

首先通过 `parseStartTag` 解析开始标签：

```

function parseStartTag () {
  const start = html.match(startTagOpen)
  if (start) {
    const match = {
      tagName: start[1],
      attrs: [],
      start: index
    }
    advance(start[0].length)
    let end, attr
    while (!(end = html.match(startTagClose)) && (attr = html.match(attribute))) {
      advance(attr[0].length)
      match.attrs.push(attr)
    }
    if (end) {
      match.unarySlash = end[1]
      advance(end[0].length)
      match.end = index
      return match
    }
  }
}

```

对于开始标签，除了标签名之外，还有一些标签相关的属性。函数先通过正则表达式 `startTagOpen` 匹配到开始标签，然后定义了 `match` 对象，接着循环去匹配开始标签中的属性并添加到 `match.attrs` 中，直到匹配的开始标签的闭合符结束。如果匹配到闭合符，则获取一元斜线符，前进到闭合符尾，并把当前索引赋值给 `match.end`。

`parseStartTag` 对开始标签解析拿到 `match` 后，紧接着会执行 `handleStartTag` 对 `match` 做处理：

```

function handleStartTag (match) {
  const tagName = match.tagName
  const unarySlash = match.unarySlash

  if (expectHTML) {
    if (lastTag === 'p' && isNonPhrasingTag(tagName)) {

```

```

        parseEndTag(lastTag)
    }
    if (canBeLeftOpenTag(tagName) && lastTag === tagName) {
        parseEndTag(tagName)
    }
}

const unary = isUnaryTag(tagName) || !unarySlash

const l = match.attrs.length
const attrs = new Array(l)
for (let i = 0; i < l; i++) {
    const args = match.attrs[i]
    if (IS_REGEX_CAPTURING_BROKEN && args[0].indexOf(''''') === -1) {
        if (args[3] === '') { delete args[3] }
        if (args[4] === '') { delete args[4] }
        if (args[5] === '') { delete args[5] }
    }
    const value = args[3] || args[4] || args[5] || ''
    const shouldDecodeNewlines = tagName === 'a' && args[1] === 'href'
        ? options.shouldDecodeNewlinesForHref
        : options.shouldDecodeNewlines
    attrs[i] = {
        name: args[1],
        value: decodeAttr(value, shouldDecodeNewlines)
    }
}

if (!unary) {
    stack.push({ tag: tagName, lowerCasedTag: tagName.toLowerCase(), attrs: attrs })
}
lastTag = tagName
}

if (options.start) {
    options.start(tagName, attrs, unary, match.start, match.end)
}
}

```

`handleStartTag` 的核心逻辑很简单，先判断开始标签是否是一元标签，类似 ``、`
` 这样，接着对 `match.attrs` 遍历并做了一些处理，最后判断如果非一元标签，则往 `stack` 里 push 一个对象，并且把 `tagName` 赋值给 `lastTag`。至于 `stack` 的作用，稍后我会介绍。

最后调用了 `options.start` 回调函数，并传入一些参数，这个回调函数的作用稍后我会详细介绍。

- 闭合标签

```

const endTagMatch = html.match(endTag)
if (endTagMatch) {
    const curIndex = index

```

```

    advance(endTagMatch[0].length)
    parseEndTag(endTagMatch[1], curIndex, index)
    continue
}

```

先通过正则 `endTag` 匹配到闭合标签，然后前进到闭合标签末尾，然后执行 `parseEndTag` 方法对闭合标签做解析。

```

function parseEndTag (tagName, start, end) {
  let pos, lowerCasedTagName
  if (start == null) start = index
  if (end == null) end = index

  if (tagName) {
    lowerCasedTagName = tagName.toLowerCase()
  }

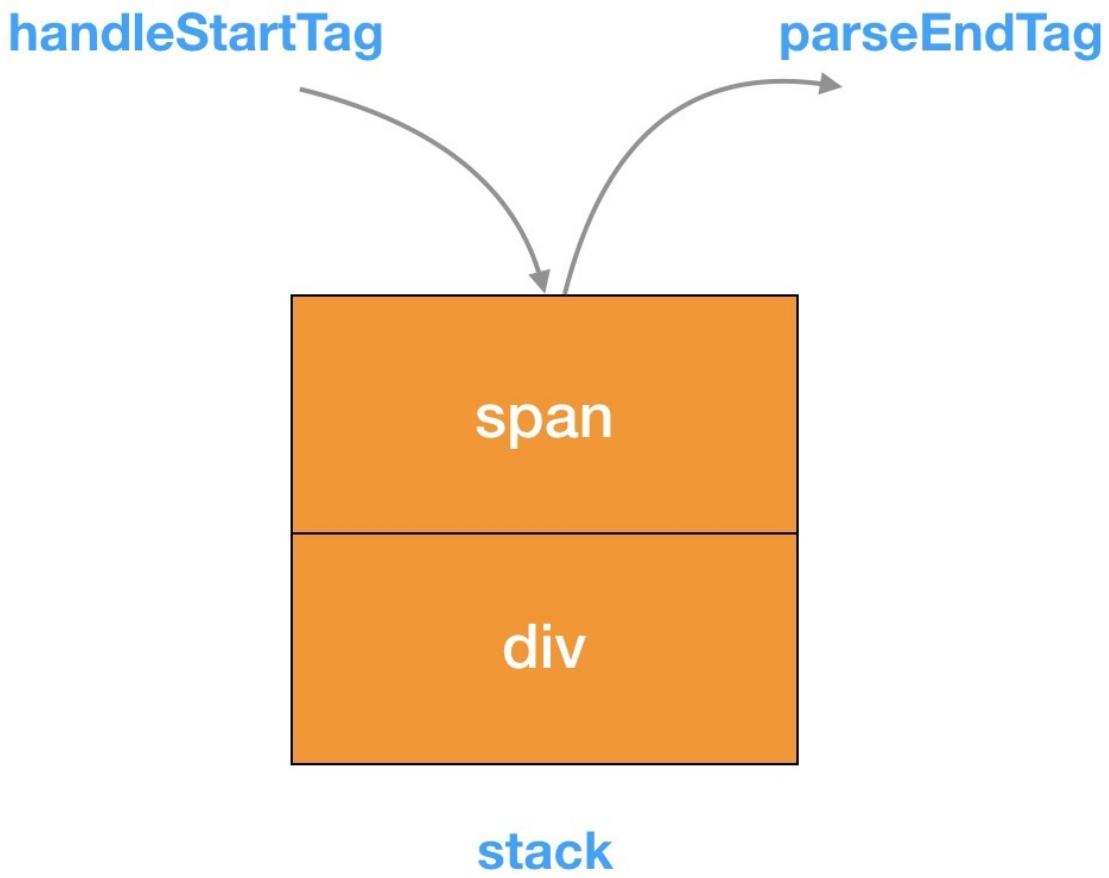
  if (tagName) {
    for (pos = stack.length - 1; pos >= 0; pos--) {
      if (stack[pos].lowerCasedTag === lowerCasedTagName) {
        break
      }
    }
  } else {
    pos = 0
  }

  if (pos >= 0) {
    for (let i = stack.length - 1; i >= pos; i--) {
      if (process.env.NODE_ENV !== 'production' &&
          (i > pos || !tagName) &&
          options.warn
      ) {
        options.warn(
          `tag <${stack[i].tag}> has no matching end tag.`
        )
      }
    }
    if (options.end) {
      options.end(stack[i].tag, start, end)
    }
  }
  stack.length = pos
  lastTag = pos && stack[pos - 1].tag
} else if (lowerCasedTagName === 'br') {
  if (options.start) {
    options.start(tagName, [], true, start, end)
  }
} else if (lowerCasedTagName === 'p') {
  if (options.start) {

```

```
        options.start(tagName, [], false, start, end)
    }
    if (options.end) {
        options.end(tagName, start, end)
    }
}
}
```

`parseEndTag` 的核心逻辑很简单，在介绍之前我们回顾一下在执行 `handleStartTag` 的时候，对于非一元标签（有 `endTag`）我们都把它构造成一个对象压入到 `stack` 中，如图所示：



那么对于闭合标签的解析，就是倒序 `stack`，找到第一个和当前 `endTag` 匹配的元素。如果是正常的标签匹配，那么 `stack` 的最后一个元素应该和当前的 `endTag` 匹配，但是考虑到如下错误情况：

```
<div><span></div>
```

这个时候当 `endTag` 为 `</div>` 的时候，从 `stack` 尾部找到的标签是 ``，就不能匹配，因此这种情况会报警告。匹配后把栈到 `pos` 位置的都弹出，并从 `stack` 尾部拿到 `lastTag`。

最后调用了 `options.end` 回调函数，并传入一些参数，这个回调函数的作用稍后我会详细介绍。

- ## ● 文本

```

let text, rest, next
if (textEnd >= 0) {
  rest = html.slice(textEnd)
  while (
    !endTag.test(rest) &&
    !startTagOpen.test(rest) &&
    !comment.test(rest) &&
    !conditionalComment.test(rest)
  ) {
    next = rest.indexOf('<', 1)
    if (next < 0) break
    textEnd += next
    rest = html.slice(textEnd)
  }
  text = html.substring(0, textEnd)
  advance(textEnd)
}

if (textEnd < 0) {
  text = html
  html = ''
}

if (options.chars && text) {
  options.chars(text)
}

```

接下来判断 `textEnd` 是否大于等于 0 的，满足则说明到从当前位置到 `textEnd` 位置都是文本，并且如果 `<` 是纯文本中的字符，就继续找到真正的文本结束的位置，然后前进到结束的位置。

再继续判断 `textEnd` 小于 0 的情况，则说明整个 `template` 解析完毕了，把剩余的 `html` 都赋值给了 `text`。

最后调用了 `options.chars` 回调函数，并传 `text` 参数，这个回调函数的作用稍后我会详细介绍。

因此，在循环解析整个 `template` 的过程中，会根据不同的情况，去执行不同的回调函数，下面来看看这些回调函数的作用。

处理开始标签

对应伪代码：

```

start (tag, attrs, unary) {
  let element = createASTElement(tag, attrs)
  processElement(element)
  treeManagement()
}

```

当解析到开始标签的时候，最后会执行 `start` 回调函数，函数主要就做 3 件事情，创建 AST 元素，处理 AST 元素，AST 树管理。下面我们来分别来看这几个过程。

- 创建 AST 元素

```
// check namespace.
// inherit parent ns if there is one
const ns = (currentParent && currentParent.ns) || platformGetTagNamespace(tag)

// handle IE svg bug
/* istanbul ignore if */
if (isIE && ns === 'svg') {
  attrs = guardIESVGBug(attrs)
}

let element: ASTElement = createASTElement(tag, attrs, currentParent)
if (ns) {
  element.ns = ns
}

export function createASTElement (
  tag: string,
  attrs: Array<Attr>,
  parent: ASTElement | void
): ASTElement {
  return {
    type: 1,
    tag,
    attrsList: attrs,
    attrsMap: makeAttrsMap(attrs),
    parent,
    children: []
  }
}
```

通过 `createASTElement` 方法去创建一个 AST 元素，并添加了 `namespace`。可以看到，每一个 AST 元素就是一个普通的 JavaScript 对象，其中，`type` 表示 AST 元素类型，`tag` 表示标签名，`attrsList` 表示属性列表，`attrsMap` 表示属性映射表，`parent` 表示父的 AST 元素，`children` 表示子 AST 元素集合。

- 处理 AST 元素

```
if (isForbiddenTag(element) && !isServerRendering()) {
  element.forbidden = true
  process.env.NODE_ENV !== 'production' && warn(
    'Templates should only be responsible for mapping the state to the ' +
    'UI. Avoid placing tags with side-effects in your templates, such as ' +
    `<${tag}>` + ', as they will not be parsed.'
)
```

```

    }

    // apply pre-transforms
    for (let i = 0; i < preTransforms.length; i++) {
        element = preTransforms[i](element, options) || element
    }

    if (!inVPre) {
        processPre(element)
        if (element.pre) {
            inVPre = true
        }
    }
    if (platformIsPreTag(element.tag)) {
        inPre = true
    }
    if (inVPre) {
        processRawAttrs(element)
    } else if (!element.processed) {
        // structural directives
        processFor(element)
        processIf(element)
        processOnce(element)
        // element-scope stuff
        processElement(element, options)
    }
}

```

首先是对模块 `preTransforms` 的调用，其实所有模块的 `preTransforms`、`transforms` 和 `postTransforms` 的定义都在 `src/platforms/web/compiler/modules` 目录中，这部分我们暂时不会介绍，之后会结合具体的例子说。接着判断 `element` 是否包含各种指令通过 `processXXX` 做相应的处理，处理的结果就是扩展 AST 元素的属性。这里我并不会一一介绍所有的指令处理，而是结合我们当前的例子，我们来看一下 `processFor` 和 `processIf`：

```

export function processFor (el: ASTElement) {
    let exp
    if ((exp = getAndRemoveAttr(el, 'v-for'))) {
        const res = parseFor(exp)
        if (res) {
            extend(el, res)
        } else if (process.env.NODE_ENV !== 'production') {
            warn(
                `Invalid v-for expression: ${exp}`
            )
        }
    }
}

export const forAliasRE = /(.*?)\s+(:in|of)\s+(.*)/
export const forIteratorRE = /,([^\{\}]*)(?:,([^\{\}]*))?\$/
```

```

const stripParensRE = /^\\(|\\)$/
export function parseFor (exp: string): ?ForParseResult {
  const inMatch = exp.match(forAliasRE)
  if (!inMatch) return
  const res = {}
  res.for = inMatch[2].trim()
  const alias = inMatch[1].trim().replace(stripParensRE, '')
  const iteratorMatch = alias.match(forIteratorRE)
  if (iteratorMatch) {
    res.alias = alias.replace(forIteratorRE, '')
    res.iterator1 = iteratorMatch[1].trim()
    if (iteratorMatch[2]) {
      res.iterator2 = iteratorMatch[2].trim()
    }
  } else {
    res.alias = alias
  }
  return res
}

```

`processFor` 就是从元素中拿到 `v-for` 指令的内容，然后分别解析出 `for`、`alias`、`iterator1`、`iterator2` 等属性的值添加到 AST 的元素上。就我们的示例 `v-for="(item,index) in data"` 而言，解析出的的 `for` 是 `data`，`alias` 是 `item`，`iterator1` 是 `index`，没有 `iterator2`。

```

function processIf (el) {
  const exp = getAndRemoveAttr(el, 'v-if')
  if (exp) {
    el.if = exp
    addIfCondition(el, {
      exp: exp,
      block: el
    })
  } else {
    if (getAndRemoveAttr(el, 'v-else') != null) {
      el.else = true
    }
    const elseif = getAndRemoveAttr(el, 'v-else-if')
    if (elseif) {
      el.elseif = elseif
    }
  }
}
export function addIfCondition (el: ASTElement, condition: ASTIfCondition) {
  if (!el.ifConditions) {
    el.ifConditions = []
  }
  el.ifConditions.push(condition)
}

```

`processIf` 就是从元素中拿 `v-if` 指令的内容，如果拿到则给 AST 元素添加 `if` 属性和 `ifConditions` 属性；否则尝试拿 `v-else` 指令及 `v-else-if` 指令的内容，如果拿到则给 AST 元素分别添加 `else` 和 `elseif` 属性。

- AST 树管理

我们在处理开始标签的时候为每一个标签创建了一个 AST 元素，在不断解析模板创建 AST 元素的时候，我们也要为它们建立父子关系，就像 DOM 元素的父子关系那样。

AST 树管理相关代码如下：

```

function checkRootConstraints (el) {
  if (process.env.NODE_ENV !== 'production') {
    if (el.tag === 'slot' || el.tag === 'template') {
      warnOnce(
        `Cannot use <${el.tag}> as component root element because it may ` +
        'contain multiple nodes.'
      )
    }
    if (el.attrsMap.hasOwnProperty('v-for')) {
      warnOnce(
        'Cannot use v-for on stateful component root element because ' +
        'it renders multiple elements.'
      )
    }
  }
}

// tree management
if (!root) {
  root = element
  checkRootConstraints(root)
} else if (!stack.length) {
  // allow root elements with v-if, v-else-if and v-else
  if (root.if && (element.elseif || element.else)) {
    checkRootConstraints(element)
    addIfCondition(root, {
      exp: element.elseif,
      block: element
    })
  }
} else if (process.env.NODE_ENV !== 'production') {
  warnOnce(
    `Component template should contain exactly one root element. ` +
    `If you are using v-if on multiple elements, ` +
    `use v-else-if to chain them instead.`
  )
}
if (currentParent && !element.forbidden) {
  if (element.elseif || element.else) {

```

```

    processIfConditions(element, currentParent)
} else if (element.slotScope) { // scoped slot
  currentParent/plain = false
  const name = element.slotTarget || '"default"'
  ;(currentParent.scopedSlots || (currentParent.scopedSlots = {}))[name] = element
} else {
  currentParent.children.push(element)
  element.parent = currentParent
}
}
if (!unary) {
  currentParent = element
  stack.push(element)
} else {
  closeElement(element)
}

```

AST 树管理的目标是构建一颗 AST 树，本质上它要维护 `root` 根节点和当前父节点 `currentParent`。为了保证元素可以正确闭合，这里也利用了 `stack` 栈的数据结构，和我们之前解析模板时用到的 `stack` 类似。

当我们在处理开始标签的时候，判断如果有 `currentParent`，会把当前 AST 元素 push 到 `currentParent.children` 中，同时把 AST 元素的 `parent` 指向 `currentParent`。

接着就是更新 `currentParent` 和 `stack`，判断当前如果不是一个一元标签，我们要把它生成的 AST 元素 push 到 `stack` 中，并且把当前的 AST 元素赋值给 `currentParent`。

`stack` 和 `currentParent` 除了在处理开始标签的时候会变化，在处理闭合标签的时候也会变化，因此整个 AST 树管理要结合闭合标签的处理逻辑看。

处理闭合标签

对应伪代码：

```

end () {
  treeManagement()
  closeElement()
}

```

当解析到闭合标签的时候，最后会执行 `end` 回调函数：

```

// remove trailing whitespace
const element = stack[stack.length - 1]
const lastNode = element.children[element.children.length - 1]
if (lastNode && lastNode.type === 3 && lastNode.text === ' ' && !inPre) {
  element.children.pop()
}
// pop stack

```

```
stack.length -= 1
currentParent = stack[stack.length - 1]
closeElement(element)
```

首先处理了尾部空格的情况，然后把 `stack` 的元素弹一个出栈，并把 `stack` 最后一个元素赋值给 `currentParent`，这样就保证了当遇到闭合标签的时候，可以正确地更新 `stack` 的长度以及 `currentParent` 的值，这样就维护了整个 AST 树。

最后执行了 `closeElement(element)`：

```
function closeElement (element) {
  // check pre state
  if (element.pre) {
    inVPre = false
  }
  if (platformIsPreTag(element.tag)) {
    inPre = false
  }
  // apply post-transforms
  for (let i = 0; i < postTransforms.length; i++) {
    postTransforms[i](element, options)
  }
}
```

`closeElement` 逻辑很简单，就是更新一下 `inVPre` 和 `inPre` 的状态，以及执行 `postTransforms` 函数，这些我们暂时都不必了解。

处理文本内容

对应伪代码：

```
chars (text: string) {
  handleText()
  createChildrenASTOfText()
}
```

除了处理开始标签和闭合标签，我们还会在解析模板的过程中去处理一些文本内容：

```
const children = currentParent.children
text = inPre || text.trim()
? isTextTag(currentParent) ? text : decodeHTMLCached(text)
// only preserve whitespace if its not right after a starting tag
: preserveWhitespace && children.length ? ' ' : ''
if (text) {
  let res
  if (!inVPre && text !== ' ' && (res = parseText(text, delimiters))) {
    children.push({
      type: 2,
```

```

        expression: res.expression,
        tokens: res.tokens,
        text
    })
} else if (text !== ' ' || !children.length || children[children.length - 1].text
!== ' ') {
    children.push({
        type: 3,
        text
    })
}
}
}

```

文本构造的 AST 元素有 2 种类型，一种是有表达式的，`type` 为 2，一种是纯文本，`type` 为 3。在我们的例子中，文本就是 `{{item}}:{{index}}`，是个表达式，通过执行 `parseText(text, delimiters)` 对文本解析，它的定义在 `src/compiler/parser/text-parse.js` 中：

```

const defaultTagRE = /\{\{((?:.|\\n)+?)\}\}/g
const regexEscapeRE = /[.*+?^${}()|[\]\\]/g

const buildRegex = cached(delimiters => {
    const open = delimiters[0].replace(regexEscapeRE, '\\$&')
    const close = delimiters[1].replace(regexEscapeRE, '\\$&')
    return new RegExp(open + '((?:.|\\n)+?)' + close, 'g')
})

export function parseText (
    text: string,
    delimiters?: [string, string]
): TextParseResult | void {
    const tagRE = delimiters ? buildRegex(delimiters) : defaultTagRE
    if (!tagRE.test(text)) {
        return
    }
    const tokens = []
    const rawTokens = []
    let lastIndex = tagRE.lastIndex = 0
    let match, index, tokenValue
    while ((match = tagRE.exec(text))) {
        index = match.index
        // push text token
        if (index > lastIndex) {
            rawTokens.push(tokenValue = text.slice(lastIndex, index))
            tokens.push(JSON.stringify(tokenValue))
        }
        // tag token
        const exp = parseFilters(match[1].trim())
        tokens.push(`_s(${exp})`)
        rawTokens.push({ '@binding': exp })
        lastIndex = index + match[0].length
    }
}

```

```

    }
    if (lastIndex < text.length) {
        rawTokens.push(tokenValue = text.slice(lastIndex))
        tokens.push(JSON.stringify(tokenValue))
    }
    return {
        expression: tokens.join('+'),
        tokens: rawTokens
    }
}

```

`parseText` 首先根据分隔符（默认是 `{}{}`）构造了文本匹配的正则表达式，然后再循环匹配文本，遇到普通文本就 push 到 `rawTokens` 和 `tokens` 中，如果是表达式就转换成 `_s(${exp})` push 到 `tokens` 中，以及转换成 `{@binding:exp}` push 到 `rawTokens` 中。

对于我们的例子 `{{item}}:{{index}}`，`tokens` 就是

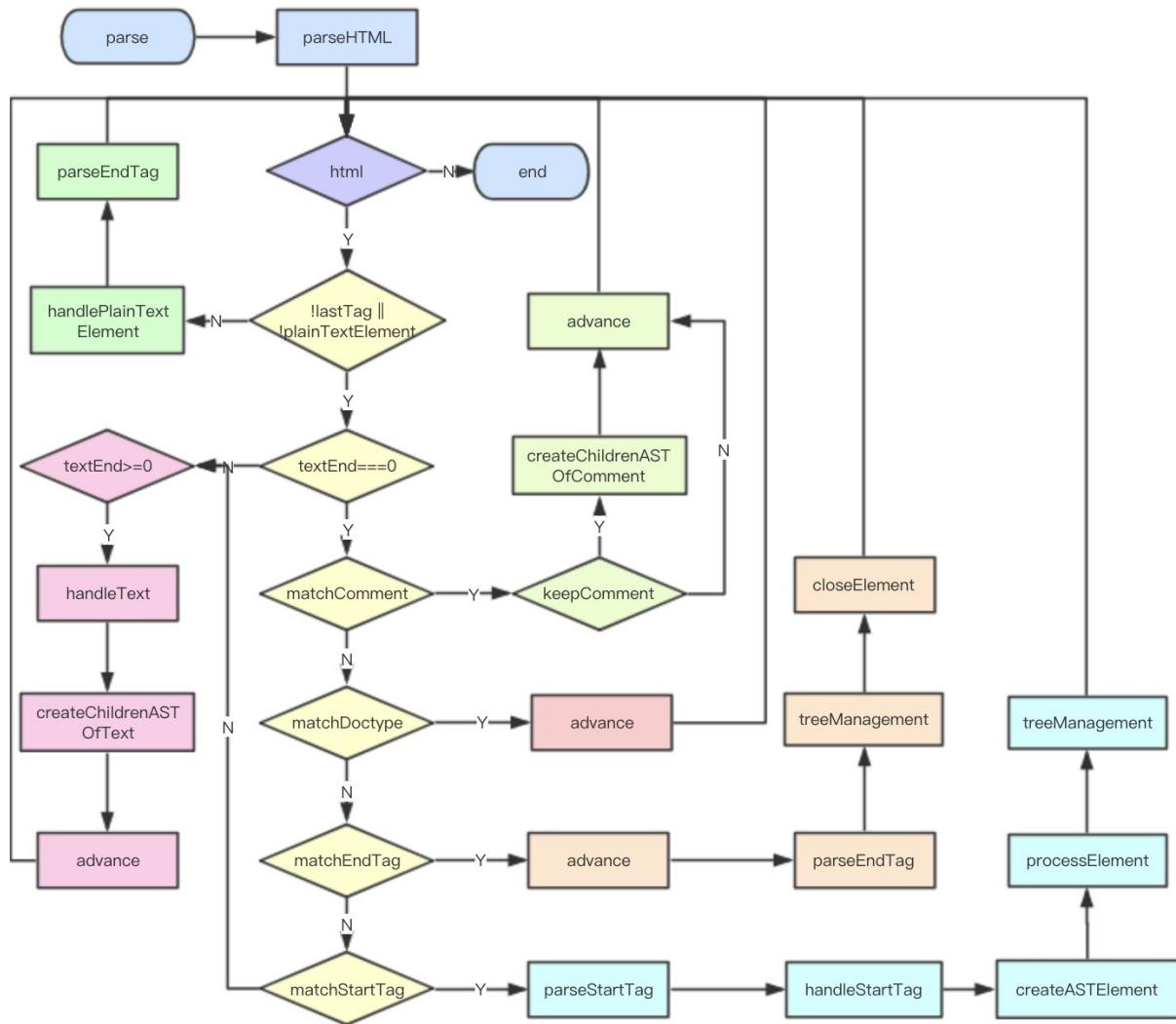
`[_s(item),':',_s(index)]`；`rawTokens` 就是 `[{@binding:'item'},':',{ '@binding':'index'}]`。那么返回的对象如下：

```

return {
    expression: '_s(item)+":"+_s(index)',
    tokens: [ {@binding:'item'},':',{ '@binding':'index'}]
}

```

流程图



总结

那么至此，`parse` 的过程就分析完了，看似复杂，但我们可以抛开细节理清它的整体流程。`parse` 的目标是把 `template` 模板字符串转换成 AST 树，它是一种用 JavaScript 对象的形式来描述整个模板。那么整个 `parse` 的过程是利用正则表达式顺序解析模板，当解析到开始标签、闭合标签、文本的时候都会分别执行对应的回调函数，来达到构造 AST 树的目的。

AST 元素节点总共有 3 种类型，`type` 为 1 表示是普通元素，为 2 表示是表达式，为 3 表示是纯文本。其实这里我觉得源码写的不够友好，这种是典型的魔术数字，如果转换成用常量表达会更利于源码阅读。

当 AST 树构造完毕，下一步就是 `optimize` 优化这颗树。

optimize

当我们的模板 `template` 经过 `parse` 过程后，会输出生成 AST 树，那么接下来我们需要对这棵树做优化，`optimize` 的逻辑是远简单于 `parse` 的逻辑，所以理解起来会轻松很多。

为什么要有优化过程，因为我们知道 Vue 是数据驱动，是响应式的，但是我们的模板并不是所有数据都是响应式的，也有很多数据是首次渲染后就永远不会变化的，那么这部分数据生成的 DOM 也不会变化，我们可以在 `patch` 的过程跳过对他们的比对。

来看一下 `optimize` 方法的定义，在 `src/compiler/optimizer.js` 中：

```
/**
 * Goal of the optimizer: walk the generated template AST tree
 * and detect sub-trees that are purely static, i.e. parts of
 * the DOM that never needs to change.
 *
 * Once we detect these sub-trees, we can:
 *
 * 1. Hoist them into constants, so that we no longer need to
 *    create fresh nodes for them on each re-render;
 * 2. Completely skip them in the patching process.
 */
export function optimize (root: ?ASTElement, options: CompilerOptions) {
  if (!root) return
  isStaticKey = genStaticKeysCached(options.staticKeys || '')
  isPlatformReservedTag = options.isReservedTag || no
  // first pass: mark all non-static nodes.
  markStatic(root)
  // second pass: mark static roots.
  markStaticRoots(root, false)
}

function genStaticKeys (keys: string): Function {
  return makeMap(
    'type,tag,attrsList,attrsMap,plain,parent,children,attrs' +
    (keys ? ',' + keys : '')
  )
}
```

我们在编译阶段可以把一些 AST 节点优化成静态节点，所以整个 `optimize` 的过程实际上就干 2 件事情，`markStatic(root)` 标记静态节点，`markStaticRoots(root, false)` 标记静态根。

标记静态节点

```
function markStatic (node: ASTNode) {
  node.static = isStatic(node)
```

```

if (node.type === 1) {
  // do not make component slot content static. this avoids
  // 1. components not able to mutate slot nodes
  // 2. static slot content fails for hot-reloading
  if (
    !isPlatformReservedTag(node.tag) &&
    node.tag !== 'slot' &&
    node.attrsMap['inline-template'] == null
  ) {
    return
  }
  for (let i = 0, l = node.children.length; i < l; i++) {
    const child = node.children[i]
    markStatic(child)
    if (!child.static) {
      node.static = false
    }
  }
  if (node.ifConditions) {
    for (let i = 0, l = node.ifConditions.length; i < l; i++) {
      const block = node.ifConditions[i].block
      markStatic(block)
      if (!block.static) {
        node.static = false
      }
    }
  }
}
}

function isStatic (node: ASTNode): boolean {
  if (node.type === 2) { // expression
    return false
  }
  if (node.type === 3) { // text
    return true
  }
  return !(node.pre || (
    !node.hasBindings && // no dynamic bindings
    !node.if && !node.for && // not v-if or v-for or v-else
    !isBuiltInTag(node.tag) && // not a built-in
    isPlatformReservedTag(node.tag) && // not a component
    !isDirectChildOfTemplateFor(node) &&
    Object.keys(node).every(isStaticKey)
  ))
}

```

首先执行 `node.static = isStatic(node)`

`isStatic` 是对一个 AST 元素节点是否是静态的判断，如果是表达式，就是非静态；如果是纯文本，就是静态；对于一个普通元素，如果有 `pre` 属性，那么它使用了 `v-pre` 指令，是静态，否则要同时满足以下条件：没有使用 `v-if`、`v-for`，没有使用其它指令（不包括 `v-once`），非内置组件，是平台保留的标签，非带有 `v-for` 的 `template` 标签的直接子节点，节点的所有属性的 `key` 都满足静态 `key`；这些都满足则这个 AST 节点是一个静态节点。

如果这个节点是一个普通元素，则遍历它的所有 `children`，递归执行 `markStatic`。因为所有的 `elseif` 和 `else` 节点都不在 `children` 中，如果节点的 `ifConditions` 不为空，则遍历 `ifConditions` 拿到所有条件中的 `block`，也就是它们对应的 AST 节点，递归执行 `markStatic`。在这些递归过程中，一旦子节点有不是 `static` 的情况，则它的父节点的 `static` 均变成 `false`。

标记静态根

```
function markStaticRoots (node: ASTNode, isInFor: boolean) {
  if (node.type === 1) {
    if (node.static || node.once) {
      node.staticInFor = isInFor
    }
    // For a node to qualify as a static root, it should have children that
    // are not just static text. Otherwise the cost of hoisting out will
    // outweigh the benefits and it's better off to just always render it fresh.
    if (node.static && node.children.length && !(node.children.length === 1 &&
      node.children[0].type === 3)) {
      node.staticRoot = true
      return
    } else {
      node.staticRoot = false
    }
    if (node.children) {
      for (let i = 0, l = node.children.length; i < l; i++) {
        markStaticRoots(node.children[i], isInFor || !node.for)
      }
    }
    if (node.ifConditions) {
      for (let i = 1, l = node.ifConditions.length; i < l; i++) {
        markStaticRoots(node.ifConditions[i].block, isInFor)
      }
    }
  }
}
```

`markStaticRoots` 第二个参数是 `isInFor`，对于已经是 `static` 的节点或者是 `v-once` 指令的节点，`node.staticInFor = isInFor`。接着就是对于 `staticRoot` 的判断逻辑，从注释中我们可以看到，对于有资格成为 `staticRoot` 的节点，除了本身是一个静态节点外，必须满足拥有

`children`，并且 `children` 不能只是一个文本节点，不然的话把它标记成静态根节点的收益就很小了。

接下来和标记静态节点的逻辑一样，遍历 `children` 以及 `ifConditions`，递归执行 `markStaticRoots`。

回归我们之前的例子，经过 `optimize` 后，AST 树变成了如下：

```
ast = {
  'type': 1,
  'tag': 'ul',
  'attrsList': [],
  'attrsMap': {
    ':class': 'bindCls',
    'class': 'list',
    'v-if': 'isShow'
  },
  'if': 'isShow',
  'ifConditions': [{{
    'exp': 'isShow',
    'block': // ul ast element
  }}],
  'parent': undefined,
  'plain': false,
  'staticClass': 'list',
  'classBinding': 'bindCls',
  'static': false,
  'staticRoot': false,
  'children': [{{
    'type': 1,
    'tag': 'li',
    'attrsList': [{{
      'name': '@click',
      'value': 'clickItem(index)'
    }}],
    'attrsMap': {
      '@click': 'clickItem(index)',
      'v-for': '(item, index) in data'
    },
    'parent': // li ast element
    'plain': false,
    'events': {
      'click': {
        'value': 'clickItem(index)'
      }
    },
    'hasBindings': true,
    'for': 'data',
    'alias': 'item',
    'iterator1': 'index',
    'static': false,
  }}]
}
```

```
'staticRoot': false,
'children': [
  'type': 2,
  'expression': '_s(item)+"_"+_s(index)'
  'text': '{{item}}:{{index}}',
  'tokens': [
    {'@binding':'item'},
    ':',
    {'@binding':'index'}
  ],
  'static': false
]
}]
}
```

我们发现每一个 AST 元素节点都多了 `staic` 属性，并且 `type` 为 1 的普通元素 AST 节点多了 `staticRoot` 属性。

总结

那么至此我们分析完了 `optimize` 的过程，就是深度遍历这个 AST 树，去检测它的每一颗子树是不是静态节点，如果是静态节点则它们生成 DOM 永远不需要改变，这对运行时对模板的更新起到极大的优化作用。

我们通过 `optimize` 我们把整个 AST 树中的每一个 AST 元素节点标记了 `static` 和 `staticRoot`，它会影响我们接下来执行代码生成的过程。

codegen

编译的最后一步就是把优化后的 AST 树转换成可执行的代码，这部分内容也比较多，我并不打算把所有的细节都讲了，了解整体流程即可。部分细节我们会在之后的章节配合一个具体 case 去详细讲。

为了方便理解，我们还是用之前的例子：

```
<ul :class="bindCls" class="list" v-if="isShow">
    <li v-for="(item, index) in data" @click="clickItem(index)">{{item}}:{{index}}</li>
</ul>
```

它经过编译，执行 `const code = generate(ast, options)`，生成的 `render` 代码串如下：

```
with(this){
    return (isShow) ?
        _c('ul', {
            staticClass: "list",
            class: bindCls
        },
        _l((data), function(item, index) {
            return _c('li', {
                on: {
                    "click": function($event) {
                        clickItem(index)
                    }
                }
            },
            [_v(_s(item) + ":" + _s(index))]
        })
    ) : _e()
}
```

这里的 `_c` 函数定义在 `src/core/instance/render.js` 中。

```
vm._c = (a, b, c, d) => createElement(vm, a, b, c, d, false)
```

而 `_l`、`_v` 定义在 `src/core/instance/render-helpers/index.js` 中：

```
export function installRenderHelpers (target: any) {
    target._o = markOnce
    target._n = toNumber
    target._s = toString
    target._l = renderList
    target._t = renderSlot
    target._q = looseEqual
```

```

target._i = looseIndexOf
target._m = renderStatic
target._f = resolveFilter
target._k = checkKeyCodes
target._b = bindObjectProps
target._v = createTextVNode
target._e = createEmptyVNode
target._u = resolveScopedSlots
target._g = bindObjectListeners
}

```

顾名思义，`_c` 就是执行 `createElement` 去创建 VNode，而 `_l` 对应 `renderList` 渲染列表；`_v` 对应 `createTextVNode` 创建文本 VNode；`_e` 对于 `createEmptyVNode` 创建空的 VNode。

在 `compileToFunctions` 中，会把这个 `render` 代码串转换成函数，它的定义在 `src/complier/to-function.js` 中：

```

const compiled = compile(template, options)
res.render = createFunction(compiled.render, fnGenErrors)

function createFunction (code, errors) {
  try {
    return new Function(code)
  } catch (err) {
    errors.push({ err, code })
    return noop
  }
}

```

实际上就是把 `render` 代码串通过 `new Function` 的方式转换成可执行的函数，赋值给 `vm.options.render`，这样当组件通过 `vm._render` 的时候，就会执行这个 `render` 函数。那么接下来我们就重点关注一下这个 `render` 代码串的生成过程。

generate

```
const code = generate(ast, options)
```

`generate` 函数的定义在 `src/compiler/codegen/index.js` 中：

```

export function generate (
  ast: ASTElement | void,
  options: CompilerOptions
): CodegenResult {
  const state = new CodegenState(options)
  const code = ast ? genElement(ast, state) : '_c("div")'
  return {
    code,
    state
  }
}

```

```

    render: `with(this){return ${code}}`,
    staticRenderFns: state.staticRenderFns
}
}

```

`generate` 函数首先通过 `genElement(ast, state)` 生成 `code`，再把 `code` 用 `with(this) {return ${code}}` 包裹起来。这里的 `state` 是 `CodegenState` 的一个实例，稍后我们在用到它的时候会介绍它。先来看一下 `genElement`：

```

export function genElement (el: ASTElement, state: CodegenState): string {
  if (el.staticRoot && !el.staticProcessed) {
    return genStatic(el, state)
  } else if (el.once && !el.onceProcessed) {
    return genOnce(el, state)
  } else if (el.for && !el.forProcessed) {
    return genFor(el, state)
  } else if (el.if && !el.ifProcessed) {
    return genIf(el, state)
  } else if (el.tag === 'template' && !el.slotTarget) {
    return genChildren(el, state) || 'void 0'
  } else if (el.tag === 'slot') {
    return genSlot(el, state)
  } else {
    // component or element
    let code
    if (el.component) {
      code = genComponent(el.component, el, state)
    } else {
      const data = el.plain ? undefined : genData(el, state)

      const children = el.inlineTemplate ? null : genChildren(el, state, true)
      code = `_c('${el.tag}')${{
        data ? `,${data}` : '' // data
      }${
        children ? `,${children}` : '' // children
      })`
    }
    // module transforms
    for (let i = 0; i < state.transforms.length; i++) {
      code = state.transforms[i](el, code)
    }
    return code
  }
}

```

基本就是判断当前 AST 元素节点的属性执行不同的代码生成函数，在我们的例子中，我们先了解一下 `genFor` 和 `genIf`。

genIf

```

export function genIf (
  el: any,
  state: CodegenState,
  altGen?: Function,
  altEmpty?: string
): string {
  el.ifProcessed = true // avoid recursion
  return genIfConditions(el.ifConditions.slice(), state, altGen, altEmpty)
}

function genIfConditions (
  conditions: ASTIfConditions,
  state: CodegenState,
  altGen?: Function,
  altEmpty?: string
): string {
  if (!conditions.length) {
    return altEmpty || '_e()'
  }

  const condition = conditions.shift()
  if (condition.exp) {
    return `(${condition.exp})?${{
      genTernaryExp(condition.block)
    }}:${
      genIfConditions(conditions, state, altGen, altEmpty)
    }`
  } else {
    return `${genTernaryExp(condition.block)}`
  }
}

// v-if with v-once should generate code like (a)?_m(0):_m(1)
function genTernaryExp (el) {
  return altGen
    ? altGen(el, state)
    : el.once
      ? genOnce(el, state)
      : genElement(el, state)
}
}

```

`genIf` 主要是通过执行 `genIfConditions`，它是依次从 `conditions` 获取第一个 `condition`，然后通过对 `condition.exp` 去生成一段三元运算符的代码，`:` 后是递归调用 `genIfConditions`，这样如果有多个 `conditions`，就生成多层三元运算逻辑。这里我们暂时不考虑 `v-once` 的情况，所以 `genTernaryExp` 最终是调用了 `genElement`。

在我们的例子中，只有一个 `condition`，`exp` 为 `isShow`，因此生成如下伪代码：

```
return (isShow) ? genElement(el, state) : _e()
```

genFor

```
export function genFor (
  el: any,
  state: CodegenState,
  altGen?: Function,
  altHelper?: string
): string {
  const exp = el.for
  const alias = el.alias
  const iterator1 = el.iterator1 ? `${el.iterator1}` : ''
  const iterator2 = el.iterator2 ? `${el.iterator2}` : ''

  if (process.env.NODE_ENV !== 'production' &&
    state.maybeComponent(el) &&
    el.tag !== 'slot' &&
    el.tag !== 'template' &&
    !el.key
  ) {
    state.warn(
      `<${el.tag} v-for="${alias} in ${exp}">: component lists rendered with ` +
      `v-for should have explicit keys. ` +
      `See https://vuejs.org/guide/list.html#key for more info.`,
      true /* tip */
    )
  }

  el.forProcessed = true // avoid recursion
  return `${altHelper || '_l'}((${exp}), ` +
    `function(${alias}${iterator1}${iterator2}){` +
    `  return ${({altGen || genElement})(el, state)}` +
    `}`)
}
```

`genFor` 的逻辑很简单，首先 AST 元素节点中获取了和 `for` 相关的一些属性，然后返回了一个代码字符串。

在我们的例子中，`exp` 是 `data`，`alias` 是 `item`，`iterator1`，因此生成如下伪代码：

```
_l((data), function(item, index) {
  return genElement(el, state)
})
```

genData & genChildren

再次回顾我们的例子，它的最外层是 `ul`，首先执行 `genIf`，它最终调用了 `genElement(el, state)` 去生成子节点，注意，这里的 `el` 仍然指向的是 `ul` 对应的 AST 节点，但是此时的 `el.ifProcessed` 为 `true`，所以命中最后一个 `else` 逻辑：

```
// component or element
let code
if (el.component) {
  code = genComponent(el.component, el, state)
} else {
  const data = el.plain ? undefined : genData(el, state)

  const children = el.inlineTemplate ? null : genChildren(el, state, true)
  code = `_c('${el.tag}')${data ? `,${data}` : ''} // data
}${children ? `,${children}` : ''} // children
)
}

// module transforms
for (let i = 0; i < state.transforms.length; i++) {
  code = state.transforms[i](el, code)
}
return code
```

这里我们只关注 2 个逻辑，`genData` 和 `genChildren`：

- `genData`

```
export function genData (el: ASTElement, state: CodegenState): string {
  let data = `

  // directives first.
  // directives may mutate the el's other properties before they are generated.
  const dirs = genDirectives(el, state)
  if (dirs) data += dirs + ','

  // key
  if (el.key) {
    data += `key:${el.key},`
  }
  // ref
  if (el.ref) {
    data += `ref:${el.ref},`
  }
  if (el.refInFor) {
    data += `refInFor:true,`
  }
  // pre
  if (el.pre) {
```

```

    data += `pre:true,`
}
// record original tag name for components using "is" attribute
if (el.component) {
  data += `tag:"${el.tag}",`
}
// module data generation functions
for (let i = 0; i < state.dataGenFns.length; i++) {
  data += state.dataGenFns[i](el)
}
// attributes
if (el.attrs) {
  data += `attrs:${genProps(el.attrs)},`
}
// DOM props
if (el.props) {
  data += `domProps:${genProps(el.props)},`
}
// event handlers
if (el.events) {
  data += `${genHandlers(el.events, false, state.warn)},`
}
if (el.nativeEvents) {
  data += `${genHandlers(el.nativeEvents, true, state.warn)},`
}
// slot target
// only for non-scoped slots
if (el.slotTarget && !el.slotScope) {
  data += `slot:${el.slotTarget},`
}
// scoped slots
if (el.scopedSlots) {
  data += `${genScopedSlots(el.scopedSlots, state)},`
}
// component v-model
if (el.model) {
  data += `model:{value:${
    el.model.value
  },callback:${
    el.model.callback
  },expression:${
    el.model.expression
  }},`
}
// inline-template
if (el.inlineTemplate) {
  const inlineTemplate = genInlineTemplate(el, state)
  if (inlineTemplate) {
    data += `${inlineTemplate},`
  }
}

```

```

data = data.replace(/,$/, '') + '}'
// v-bind data wrap
if (el.wrapData) {
  data = el.wrapData(data)
}
// v-on data wrap
if (el.wrapListeners) {
  data = el.wrapListeners(data)
}
return data
}

```

`genData` 函数就是根据 AST 元素节点的属性构造出一个 `data` 对象字符串，这个在后面创建 VNode 的时候的时候会作为参数传入。

之前我们提到了 `CodegenState` 的实例 `state`，这里有一段关于 `state` 的逻辑：

```

for (let i = 0; i < state.dataGenFns.length; i++) {
  data += state.dataGenFns[i](el)
}

```

`state.dataGenFns` 的初始化在它的构造器中。

```

export class CodegenState {
  constructor (options: CompilerOptions) {
    // ...
    this.dataGenFns = pluckModuleFunction(options.modules, 'genData')
    // ...
  }
}

```

实际上就是获取所有 `modules` 中的 `genData` 函数，其中，`class module` 和 `style module` 定义了 `genData` 函数。比如定义在 `src/platforms/web/compiler/modules/class.js` 中的 `genData` 方法：

```

function genData (el: ASTElement): string {
  let data = ''
  if (el.staticClass) {
    data += `staticClass:${el.staticClass},`
  }
  if (el.classBinding) {
    data += `class:${el.classBinding},`
  }
  return data
}

```

在我们的例子中，`ul` AST 元素节点定义了 `el.staticClass` 和 `el.classBinding`，因此最终生成的 `data` 字符串如下：

```
{
  staticClass: "list",
  class: bindCls
}
```

- genChildren

接下来我们再来看一下 `genChildren`，它的定义在 `src/compiler/codegen/index.js` 中：

```
export function genChildren (
  el: ASTElement,
  state: CodegenState,
  checkSkip?: boolean,
  altGenElement?: Function,
  altGenNode?: Function
): string | void {
  const children = el.children
  if (children.length) {
    const el: any = children[0]
    if (children.length === 1 &&
        el.for &&
        el.tag !== 'template' &&
        el.tag !== 'slot')
    ) {
      return (altGenElement || genElement)(el, state)
    }
    const normalizationType = checkSkip
      ? getNormalizationType(children, state.maybeComponent)
      : 0
    const gen = altGenNode || genNode
    return `[${children.map(c => gen(c, state)).join(',')}]${{
      normalizationType ? `,${normalizationType}` : ''
    }}`
  }
}
```

在我们的例子中，`li` AST 元素节点是 `ul` AST 元素节点的 `children` 之一，满足 `(children.length === 1 && el.for && el.tag !== 'template' && el.tag !== 'slot')` 条件，因此通过 `genElement(el, state)` 生成 `li` AST 元素节点的代码，也就回到了我们之前调用 `genFor` 生成的代码，把它们拼在一起生成的伪代码如下：

```
return (isShow) ?
  _c('ul', {
    staticClass: "list",
    class: bindCls
```

```

},
_l((data), function(item, index) {
    return genElement(el, state)
})
) : _e()

```

在我们的例子中，在执行 `genElement(el, state)` 的时候，`el` 还是 `li` AST 元素节点，`el.forProcessed` 已为 `true`，所以会继续执行 `genData` 和 `genChildren` 的逻辑。由于 `el.events` 不为空，在执行 `genData` 的时候，会执行如下逻辑：

```

if (el.events) {
    data += `${genHandlers(el.events, false, state.warn)},` 
}

```

`genHandlers` 的定义在 `src/compiler/codegen/events.js` 中：

```

export function genHandlers (
    events: ASTElementHandlers,
    isNative: boolean,
    warn: Function
): string {
    let res = isNative ? 'nativeOn:{' : 'on:{'
    for (const name in events) {
        res += `${name}: ${genHandler(name, events[name])},`
    }
    return res.slice(0, -1) + '}'
}

```

`genHandler` 的逻辑就不介绍了，很大部分都是对修饰符 `modifier` 的处理，感兴趣同学可以自己看，对于我们的例子，它最终 `genData` 生成的 `data` 字符串如下：

```
{
  on: {
    "click": function($event) {
      clickItem(index)
    }
  }
}
```

`genChildren` 的时候，会执行到如下逻辑：

```

export function genChildren (
    el: ASTElement,
    state: CodegenState,
    checkSkip?: boolean,
    altGenElement?: Function,
    altGenNode?: Function
)

```

```

): string | void {
  // ...
  const normalizationType = checkSkip
    ? getNormalizationType(children, state.maybeComponent)
    : 0
  const gen = altGenNode || genNode
  return `[${
    children.map(c => gen(c, state)).join(',')
  }]${normalizationType ? `,${normalizationType}` : ''}
}
}

function genNode (node: ASTNode, state: CodegenState): string {
  if (node.type === 1) {
    return genElement(node, state)
  } if (node.type === 3 && node.isComment) {
    return genComment(node)
  } else {
    return genText(node)
  }
}

```

`genChildren` 的就是遍历 `children`，然后执行 `genNode` 方法，根据不同的 `type` 执行具体的方法。在我们的例子中，`li` AST 元素节点的 `children` 是 `type` 为 2 的表达式 AST 元素节点，那么会执行到 `genText(node)` 逻辑。

```

export function genText (text: ASTText | ASTExpression): string {
  return `_v(${text.type === 2
    ? text.expression
    : transformSpecialNewlines(JSON.stringify(text.text))})`
}

```

因此在我们的例子中，`genChildren` 生成的代码串如下：

```
[_v(_s(item) + ":" + _s(index))]
```

和之前拼在一起，最终生成的 `code` 如下：

```

return (isShow) ?
  _c('ul', {
    staticClass: "list",
    class: bindCls
  },
  _l((data), function(item, index) {
    return _c('li', {
      on: {
        "click": function($event) {
          clickItem(index)
        }
      }
    })
  })
}

```

```
        }
    }
},
[_v(_s(item) + ":" + _s(index))])
})
) : _e()
```

总结

这一节通过例子配合解析，我们对从 `ast -> code` 这一步有了一些了解，编译后生成的代码就是在运行时执行的代码。由于 `genCode` 的内容有很多，所以我对大家的建议是没必要把所有的细节都一次性看完，我们应该根据具体一个 case，走完一条主线即可。

在之后的章节我们会对 `slot` 的实现做解析，我们会重新复习编译的章节，针对具体问题做具体分析，有利于我们排除干扰，对编译过程的学习有更深入的理解。

扩展

前面几章我们分析了 Vue 的核心以及编译过程，除此之外，Vue 还提供了很多好用的 feature 如 `event`、`v-model`、`slot`、`keep-alive`、`transition` 等等。对他们的理解有助于我们在平时开发中更好地应用这些 feature，即使出现 bug 我们也可以很从容地应对。

这一章是一个可扩展的章节，除了已分析的这些 feature 外，未来我们可能会扩展更多的内容。

event

我们平时开发工作中，处理组件间的通讯，原生的交互，都离不开事件。对于一个组件元素，我们不仅仅可以绑定原生的 DOM 事件，还可以绑定自定义事件，非常灵活和方便。那么接下来我们从源码角度来看看它的实现原理。

为了更加直观，我们通过一个例子来分析它的实现：

```
let Child = {
  template: '<button @click="clickHandler($event)">' +
  'click me' +
  '</button>',
  methods: {
    clickHandler(e) {
      console.log('Button clicked!', e)
      this.$emit('select')
    }
  }
}

let vm = new Vue({
  el: '#app',
  template: '<div>' +
  '<child @select="selectHandler" @click.native.prevent="clickHandler"></child>' +
  '</div>',
  methods: {
    clickHandler() {
      console.log('Child clicked!')
    },
    selectHandler() {
      console.log('Child select!')
    }
  },
  components: {
    Child
  }
})
```

编译

先从编译阶段开始看起，在 `parse` 阶段，会执行 `processAttrs` 方法，它的定义在 `src/compiler/parser/index.js` 中：

```
export const onRE = /^@|^v-on:/;
export const dirRE = /^v-|@|:/;
export const bindRE = /^:|^v-bind:/;
```

```

function processAttrs (el) {
  const list = el.attrsList
  let i, l, name, rawName, value, modifiers, isProp
  for (i = 0, l = list.length; i < l; i++) {
    name = rawName = list[i].name
    value = list[i].value
    if (dirRE.test(name)) {
      el.hasBindings = true
      modifiers = parseModifiers(name)
      if (modifiers) {
        name = name.replace(modifierRE, '')
      }
      if (bindRE.test(name)) {
        // ...
      } else if (onRE.test(name)) {
        name = name.replace(onRE, '')
        addHandler(el, name, value, modifiers, false, warn)
      } else {
        // ...
      }
    } else {
      // ...
    }
  }
}

function parseModifiers (name: string): Object | void {
  const match = name.match(modifierRE)
  if (match) {
    const ret = {}
    match.forEach(m => { ret[m.slice(1)] = true })
    return ret
  }
}

```

在对标签属性的处理过程中，判断如果是指令，首先通过 `parseModifiers` 解析出修饰符，然后判断如果事件的指令，则执行 `addHandler(el, name, value, modifiers, false, warn)` 方法，它的定义在 `src/compiler/helpers.js` 中：

```

export function addHandler (
  el: ASTElement,
  name: string,
  value: string,
  modifiers: ?ASTModifiers,
  important?: boolean,
  warn?: Function
) {
  modifiers = modifiers || emptyObject
  // warn prevent and passive modifier
  /* istanbul ignore if */

```

```
if (
  process.env.NODE_ENV !== 'production' && warn &&
  modifiers.prevent && modifiers.passive
) {
  warn(
    'passive and prevent can\'t be used together. ' +
    'Passive handler can\'t prevent default event.'
  )
}

// check capture modifier
if (modifiers.capture) {
  delete modifiers.capture
  name = '!' + name // mark the event as captured
}
if (modifiers.once) {
  delete modifiers.once
  name = '~' + name // mark the event as once
}
/* istanbul ignore if */
if (modifiers.passive) {
  delete modifiers.passive
  name = '&' + name // mark the event as passive
}

// normalize click.right and click.middle since they don't actually fire
// this is technically browser-specific, but at least for now browsers are
// the only target envs that have right/middle clicks.
if (name === 'click') {
  if (modifiers.right) {
    name = 'contextmenu'
    delete modifiers.right
  } else if (modifiers.middle) {
    name = 'mouseup'
  }
}

let events
if (modifiers.native) {
  delete modifiers.native
  events = el.nativeEvents || (el.nativeEvents = {})
} else {
  events = el.events || (el.events = {})
}

const newHandler: any = {
  value: value.trim()
}
if (modifiers !== emptyObject) {
  newHandler.modifiers = modifiers
}
```

```

const handlers = events[name]
/* istanbul ignore if */
if (Array.isArray(handlers)) {
  important ? handlers.unshift(newHandler) : handlers.push(newHandler)
} else if (handlers) {
  events[name] = important ? [newHandler, handlers] : [handlers, newHandler]
} else {
  events[name] = newHandler
}

el.plain = false
}

```

`addHandler` 函数看起来长，实际上就做了 3 件事情，首先根据 `modifier` 修饰符对事件名 `name` 做处理，接着根据 `modifier.native` 判断是一个纯原生事件还是普通事件，分别对应 `el.nativeEvents` 和 `el.events`，最后按照 `name` 对事件做归类，并把回调函数的字符串保留到对应的事件中。

在我们的例子中，父组件的 `child` 节点生成的 `el.events` 和 `el.nativeEvents` 如下：

```

el.events = {
  select: {
    value: 'selectHandler'
  }
}

el.nativeEvents = {
  click: {
    value: 'clickHandler',
    modifiers: {
      prevent: true
    }
  }
}

```

子组件的 `button` 节点生成的 `el.events` 如下：

```

el.events = {
  click: {
    value: 'clickHandler($event)'
  }
}

```

然后在 `codegen` 的阶段，会在 `genData` 函数中根据 AST 元素节点上的 `events` 和 `nativeEvents` 生成 `data` 数据，它的定义在 `src/compiler/codegen/index.js` 中：

```
export function genData (el: ASElement, state: CodegenState): string {
```

```

let data = '{'
// ...
if (el.events) {
  data += `${genHandlers(el.events, false, state.warn)},` 
}
if (el.nativeEvents) {
  data += `${genHandlers(el.nativeEvents, true, state.warn)},` 
}
// ...
return data
}

```

对于这两个属性，会调用 `genHandlers` 函数，定义在 `src/compiler/codegen/events.js` 中：

```

export function genHandlers (
  events: ASTElementHandlers,
  isNative: boolean,
  warn: Function
): string {
  let res = isNative ? 'nativeOn:{' : 'on:{'
  for (const name in events) {
    res += `${name}: ${genHandler(name, events[name])},` 
  }
  return res.slice(0, -1) + '}'
}

const fnExpRE = /^\s*([\w$_]+|(\[^)]*?)\s*=>|^function\s*\(/
const simplePathRE = /^\s*[A-Za-z$_][\w$]*(?:\.[A-Za-z$_][\w$]*|\['.*?']|\[".*?"'])|\d+|[A-Za-z$_][\w$]*\s*$/ 
function genHandler (
  name: string,
  handler: ASTElementHandler | Array<ASTElementHandler>
): string {
  if (!handler) {
    return 'function(){}
  }

  if (Array.isArray(handler)) {
    return `[${handler.map(handler => genHandler(name, handler)).join(',')}]` 
  }

  const isMethodPath = simplePathRE.test(handler.value)
  const isFunctionExpression = fnExpRE.test(handler.value)

  if (!handler.modifiers) {
    if (isMethodPath || isFunctionExpression) {
      return handler.value
    }
    /* istanbul ignore if */
    if (__WEEEX__ && handler.params) {

```

```

        return genWeexHandler(handler.params, handler.value)
    }
    return `function($event){${handler.value}}` // inline statement
} else {
  let code = ''
  let genModifierCode = ''
  const keys = []
  for (const key in handler.modifiers) {
    if (modifierCode[key]) {
      genModifierCode += modifierCode[key]
      // left/right
      if (keyCodes[key]) {
        keys.push(key)
      }
    } else if (key === 'exact') {
      const modifiers: ASTModifiers = (handler.modifiers: any)
      genModifierCode += genGuard(
        ['ctrl', 'shift', 'alt', 'meta']
        .filter(keyModifier => !modifiers[keyModifier])
        .map(keyModifier => `${$event.${keyModifier}Key}`)
        .join('||')
      )
    } else {
      keys.push(key)
    }
  }
  if (keys.length) {
    code += genKeyFilter(keys)
  }
  // Make sure modifiers like prevent and stop get executed after key filtering
  if (genModifierCode) {
    code += genModifierCode
  }
  const handlerCode = isMethodPath
    ? `return ${handler.value}(${$event})`
    : isFunctionExpression
    ? `return (${handler.value})(${$event})`
    : handler.value
  /* istanbul ignore if */
  if (__WEEX__ && handler.params) {
    return genWeexHandler(handler.params, code + handlerCode)
  }
  return `function($event){${code}${handlerCode}}`
}
}

```

`genHandlers` 方法遍历事件对象 `events`，对同一个事件名称的事件调用 `genHandler(name, events[name])` 方法，它的内容看起来多，但实际上逻辑很简单，首先先判断如果 `handler` 是一个数组，就遍历它然后递归调用 `genHandler` 方法并拼接结果，然后判断 `handler.value` 是一个函

数的调用路径还是一个函数表达式，接着对 `modifiers` 做判断，对于没有 `modifiers` 的情况，就根据 `handler.value` 不同情况处理，要么直接返回，要么返回一个函数包裹的表达式；对于有 `modifiers` 的情况，则对各种不同的 `modifier` 情况做不同处理，添加相应的代码串。

那么对于我们的例子而言，父组件生成的 `data` 串为：

```
{
  on: {"select": selectHandler},
  nativeOn: {"click": function($event) {
    $event.preventDefault();
    return clickHandler($event)
  }}
}
```

子组件生成的 `data` 串为：

```
{
  on: {"click": function($event) {
    clickHandler($event)
  }}
}
```

那么到这里，编译部分完了，接下来我们来看一下运行时部分是如何实现的。其实 Vue 的事件有 2 种，一种是原生 DOM 事件，一种是用户自定义事件，我们分别来看。

DOM 事件

还记得我们之前在 `patch` 的时候执行各种 `module` 的钩子函数吗，当时这部分是略过的，我们之前只分析了 DOM 是如何渲染的，而 DOM 元素相关的属性、样式、事件等都是通过这些 `module` 的钩子函数完成设置的。

所有和 web 相关的 `module` 都定义在 `src/platforms/web/runtime/modules` 目录下，我们这次只关注目录下的 `events.js` 即可。

在 `patch` 过程中的创建阶段和更新阶段都会执行 `updateDOMListeners`：

```
let target: any
function updateDOMListeners (oldVnode: VNodeWithData, vnode: VNodeWithData) {
  if (isUndef(oldVnode.data.on) && isUndef(vnode.data.on)) {
    return
  }
  const on = vnode.data.on || {}
  const oldOn = oldVnode.data.on || {}
  target = vnode.elm
  normalizeEvents(on)
  updateListeners(on, oldOn, add, remove, vnode.context)
```

```
    target = undefined
}
```

首先获取 `vnode.data.on`，这是我们之前的生成的 `data` 中对应的事件对象，`target` 是当前 `vnode` 对应的 DOM 对象，`normalizeEvents` 主要是对 `v-model` 相关的处理，我们之后分析 `v-model` 的时候会介绍，接着调用 `updateListeners(on, oldOn, add, remove, vnode.context)` 方法，它的定义在 `src/core/vdom/helpers/update-listeners.js` 中：

```
export function updateListeners (
  on: Object,
  oldOn: Object,
  add: Function,
  remove: Function,
  vm: Component
) {
  let name, def, cur, old, event
  for (name in on) {
    def = cur = on[name]
    old = oldOn[name]
    event = normalizeEvent(name)
    /* istanbul ignore if */
    if (__WEEEX__ && isPlainObject(def)) {
      cur = def.handler
      event.params = def.params
    }
    if (isUndef(cur)) {
      process.env.NODE_ENV !== 'production' && warn(
        `Invalid handler for event "${event.name}": got ` + String(cur),
        vm
      )
    } else if (isUndef(old)) {
      if (isUndef(cur.fns)) {
        cur = on[name] = createFnInvoker(cur)
      }
      add(event.name, cur, event.once, event.capture, event.passive, event.params)
    } else if (cur !== old) {
      old.fns = cur
      on[name] = old
    }
  }
  for (name in oldOn) {
    if (isUndef(on[name])) {
      event = normalizeEvent(name)
      remove(event.name, oldOn[name], event.capture)
    }
  }
}
```

`updateListeners` 的逻辑很简单，遍历 `on` 去添加事件监听，遍历 `oldOn` 去移除事件监听，关于监听和移除事件的方法都是外部传入的，因为它既处理原生 DOM 事件的添加删除，也处理自定义事件的添加删除。

对于 `on` 的遍历，首先获得每一个事件名，然后做 `normalizeEvent` 的处理：

```
const normalizeEvent = cached((name: string): {
  name: string,
  once: boolean,
  capture: boolean,
  passive: boolean,
  handler?: Function,
  params?: Array<any>
}) => {
  const passive = name.charAt(0) === '&'
  name = passive ? name.slice(1) : name
  const once = name.charAt(0) === '~' // Prefixed last, checked first
  name = once ? name.slice(1) : name
  const capture = name.charAt(0) === '!'
  name = capture ? name.slice(1) : name
  return {
    name,
    once,
    capture,
    passive
  }
})
```

根据我们的事件名的一些特殊标识（之前在 `addHandler` 的时候添加上的）区分出这个事件是否有 `once`、`capture`、`passive` 等修饰符。

处理完事件名后，又对事件回调函数做处理，对于第一次，满足 `isUndef(old)` 并且 `isUndef(cur.fns)`，会执行 `cur = on[name] = createFnInvoker(cur)` 方法去创建一个回调函数，然后在执行 `add(event.name, cur, event.once, event.capture, event.passive, event.params)` 完成一次事件绑定。我们先看一下 `createFnInvoker` 的实现：

```
export function createFnInvoker (fns: Function | Array<Function>): Function {
  function invoker () {
    const fns = invoker.fns
    if (Array.isArray(fns)) {
      const cloned = fns.slice()
      for (let i = 0; i < cloned.length; i++) {
        cloned[i].apply(null, arguments)
      }
    } else {
      return fns.apply(null, arguments)
    }
  }
  invoker.fns = fns
```

```
    return invoker
}
```

这里定义了 `invoker` 方法并返回，由于一个事件可能会对应多个回调函数，所以这里做了数组的判断，多个回调函数就依次调用。注意最后的赋值逻辑，`invoker.fns = fns`，每一次执行 `invoker` 函数都是从 `invoker.fns` 里取执行的回调函数，回到 `updateListeners`，当我们第二次执行该函数的时候，判断如果 `cur !== old`，那么只需要更改 `old.fns = cur` 把之前绑定的 `invoker.fns` 赋值为新的回调函数即可，并且通过 `on[name] = old` 保留引用关系，这样就保证了事件回调只添加一次，之后仅仅去修改它的回调函数的引用。

`updateListeners` 函数的最后遍历 `oldOn` 拿到事件名称，判断如果满足 `isUndef(on[name])`，则执行 `remove(event.name, oldOn[name], event.capture)` 去移除事件回调。

了解了 `updateListeners` 的实现后，我们来看一下在原生 DOM 事件中真正添加回调和移除回调函数的实现，它们的定义都在 `src/platforms/web/runtime/modules/event.js` 中：

```
function add (
  event: string,
  handler: Function,
  once: boolean,
  capture: boolean,
  passive: boolean
) {
  handler = withMacroTask(handler)
  if (once) handler = createOnceHandler(handler, event, capture)
  target.addEventListener(
    event,
    handler,
    supportsPassive
      ? { capture, passive }
      : capture
  )
}

function remove (
  event: string,
  handler: Function,
  capture: boolean,
  _target?: HTMLElement
) {
  (_target || target).removeEventListener(
    event,
    handler._withTask || handler,
    capture
  )
}
```

`add` 和 `remove` 的逻辑很简单，就是实际上调用原生 `addEventListener` 和 `removeEventListener`，并根据参数传递一些配置，注意这里的 `handler` 会用 `withMacroTask(handler)` 包裹一下，它的定义在 `src/core/util/next-tick.js` 中：

```
export function withMacroTask (fn: Function): Function {
  return fn._withTask || (fn._withTask = function () {
    useMacroTask = true
    const res = fn.apply(null, arguments)
    useMacroTask = false
    return res
  })
}
```

实际上就是强制在 DOM 事件的回调函数执行期间如果修改了数据，那么这些数据更改推入的队列会被当做 `macroTask` 在 `nextTick` 后执行。

自定义事件

除了原生 DOM 事件，Vue 还支持了自定义事件，并且自定义事件只能作用在组件上，如果在组件上使用原生事件，需要加 `.native` 修饰符，普通元素上使用 `.native` 修饰符无效，接下来我们就来分析它的实现。

在 `render` 阶段，如果是一个组件节点，则通过 `createComponent` 创建一个组件 `vnode`，我们再来看看这个方法，定义在 `src/core/vdom/create-component.js` 中：

```
export function createComponent (
  Ctor: Class<Component> | Function | Object | void,
  data: ?VNodeData,
  context: Component,
  children: ?Array<VNode>,
  tag?: string
): VNode | Array<VNode> | void {
  // ...
  const listeners = data.on

  data.on = data.nativeOn

  // ...
  const name = Ctor.options.name || tag
  const vnode = new VNode(
    `vue-component-${Ctor.cid}${name ? `-${name}` : ''}`,
    data, undefined, undefined, undefined, context,
    { Ctor, propsData, listeners, tag, children },
    asyncFactory
  )

  return vnode
}
```

我们只关注事件相关的逻辑，可以看到，它把 `data.on` 赋值给了 `listeners`，把 `data.nativeOn` 赋值给了 `data.on`，这样所有的原生 DOM 事件处理跟我们刚才介绍的一样，它是在当前组件环境中处理的。而对于自定义事件，我们把 `listeners` 作为 `vnode` 的 `componentOptions` 传入，它是在子组件初始化阶段中处理的，所以它的处理环境是子组件。

然后在子组件的初始化的时候，会执行 `initInternalComponent` 方法，它的定义在 `src/core/instance/init.js` 中：

```
export function initInternalComponent (vm: Component, options: InternalComponentOptions) {
  const opts = vm.$options = Object.create(vm.constructor.options)
  // ...
  const vnodeComponentOptions = parentVnode.componentOptions

  opts._parentListeners = vnodeComponentOptions.listeners
  // ...
}
```

这里拿到了父组件传入的 `listeners`，然后在执行 `initEvents` 的过程中，会处理这个 `listeners`，定义在 `src/core/instance/events.js` 中：

```
export function initEvents (vm: Component) {
  vm._events = Object.create(null)
  vm._hasHookEvent = false
  // init parent attached events
  const listeners = vm.$options._parentListeners
  if (listeners) {
    updateComponentListeners(vm, listeners)
  }
}
```

拿到 `listeners` 后，执行 `updateComponentListeners(vm, listeners)` 方法：

```
let target: any
export function updateComponentListeners (
  vm: Component,
  listeners: Object,
  oldListeners: ?Object
) {
  target = vm
  updateListeners(listeners, oldListeners || {}, add, remove, vm)
  target = undefined
}
```

`updateListeners` 我们之前介绍过，所以对于自定义事件和原生 DOM 事件处理的差异就在事件添加和删除的实现上，来看一下自定义事件 `add` 和 `remove` 的实现：

```

function add (event, fn, once) {
  if (once) {
    target.$once(event, fn)
  } else {
    target.$on(event, fn)
  }
}

function remove (event, fn) {
  target.$off(event, fn)
}

```

实际上是利用 Vue 定义的事件中心，简单分析一下它的实现：

```

export function eventsMixin (Vue: Class<Component>) {
  const hookRE = /^hook:/
  Vue.prototype.$on = function (event: string | Array<string>, fn: Function): Component {
    const vm: Component = this
    if (Array.isArray(event)) {
      for (let i = 0, l = event.length; i < l; i++) {
        this.$on(event[i], fn)
      }
    } else {
      (vm._events[event] || (vm._events[event] = [])).push(fn)
      // optimize hook:event cost by using a boolean flag marked at registration
      // instead of a hash lookup
      if (hookRE.test(event)) {
        vm._hasHookEvent = true
      }
    }
    return vm
  }

  Vue.prototype.$once = function (event: string, fn: Function): Component {
    const vm: Component = this
    function on () {
      vm.$off(event, on)
      fn.apply(vm, arguments)
    }
    on.fn = fn
    vm.$on(event, on)
    return vm
  }

  Vue.prototype.$off = function (event?: string | Array<string>, fn?: Function): Component {
    const vm: Component = this
    // all
    if (!arguments.length) {

```

```

    vm._events = Object.create(null)
    return vm
}
// array of events
if (Array.isArray(event)) {
  for (let i = 0, l = event.length; i < l; i++) {
    this.$off(event[i], fn)
  }
  return vm
}
// specific event
const cbs = vm._events[event]
if (!cbs) {
  return vm
}
if (!fn) {
  vm._events[event] = null
  return vm
}
if (fn) {
  // specific handler
  let cb
  let i = cbs.length
  while (i--) {
    cb = cbs[i]
    if (cb === fn || cb.fn === fn) {
      cbs.splice(i, 1)
      break
    }
  }
}
return vm
}

Vue.prototype.$emit = function (event: string): Component {
  const vm: Component = this
  if (process.env.NODE_ENV !== 'production') {
    const lowerCaseEvent = event.toLowerCase()
    if (lowerCaseEvent !== event && vm._events[lowerCaseEvent]) {
      tip(
        `Event "${lowerCaseEvent}" is emitted in component ` +
        `${formatComponentName(vm)} but the handler is registered for "${event}".
      +
        `Note that HTML attributes are case-insensitive and you cannot use ` +
        `v-on to listen to camelCase events when using in-DOM templates. ` +
        `You should probably use "${hyphenate(event)}" instead of "${event}".
      )
    }
  }
  let cbs = vm._events[event]
  if (cbs) {

```

```

        cbs = cbs.length > 1 ? toArray(cbs) : cbs
        const args = toArray(arguments, 1)
        for (let i = 0, l = cbs.length; i < l; i++) {
            try {
                cbs[i].apply(vm, args)
            } catch (e) {
                handleError(e, vm, `event handler for "${event}"`)
            }
        }
    }
    return vm
}
}

```

非常经典的事件中心的实现，把所有的事件用 `vm._events` 存储起来，当执行 `vm.$on(event, fn)` 的时候，按事件的名称 `event` 把回调函数 `fn` 存储起来 `vm._events[event].push(fn)`。当执行 `vm.$emit(event)` 的时候，根据事件名 `event` 找到所有的回调函数 `let cbs = vm._events[event]`，然后遍历执行所有的回调函数。当执行 `vm.$off(event, fn)` 的时候会移除指定事件名 `event` 和指定的 `fn`。当执行 `vm.$once(event, fn)` 的时候，内部就是执行 `vm.$on`，并且当回调函数执行一次后再通过 `vm.$off` 移除事件的回调，这样就确保了回调函数只执行一次。

所以对于用户自定义的事件添加和删除就是利用了这几个事件中心的 API。需要注意的一点，`vm.$emit` 是给当前的 `vm` 上派发的实例，之所以我们常用它做父子组件通讯，是因为它的回调函数的定义是在父组件中，对于我们这个例子而言，当子组件的 `button` 被点击了，它通过 `this.$emit('select')` 派发事件，那么子组件的实例就监听到了这个 `select` 事件，并执行它的回调函数——定义在父组件中的 `selectHandler` 方法，这样就相当于完成了一次父子组件的通讯。

总结

那么至此我们对 Vue 的事件实现有了进一步的了解，Vue 支持 2 种事件类型，原生 DOM 事件和自定义事件，它们主要的区别在于添加和删除事件的方式不一样，并且自定义事件的派发是往当前实例上派发，但是可以利用在父组件环境定义回调函数来实现父子组件的通讯。另外要注意一点，只有组件节点才可以添加自定义事件，并且添加原生 DOM 事件需要使用 `native` 修饰符；而普通元素使用 `.native` 修饰符是没有作用的，也只能添加原生 DOM 事件。

v-model

很多同学在理解 Vue 的时候都把 Vue 的数据响应原理理解为双向绑定，但实际上这是不准确的，我们之前提到的数据响应，都是通过数据的改变去驱动 DOM 视图的变化，而双向绑定除了数据驱动 DOM 外，DOM 的变化反过来影响数据，是一个双向关系，在 Vue 中，我们可以通过 `v-model` 来实现双向绑定。

`v-model` 即可以作用在普通表单元素上，又可以作用在组件上，它其实是一个语法糖，接下来我们就来分析 `v-model` 的实现原理。

表单元素

为了更加直观，我们还是结合示例来分析：

```
let vm = new Vue({
  el: '#app',
  template: '<div>' +
    '<input v-model="message" placeholder="edit me">' +
    '<p>Message is: {{ message }}</p>' +
    '</div>',
  data() {
    return {
      message: ''
    }
  }
})
```

这是一个非常简单的 demo，我们在 `input` 元素上设置了 `v-model` 属性，绑定了 `message`，当我们在 `input` 上输入了内容，`message` 也会同步变化。接下来我们就来分析 Vue 是如何实现这一效果的，其实非常简单。

也是先从编译阶段分析，首先是 `parse` 阶段，`v-model` 被当做普通的指令解析到 `el.directives` 中，然后在 `codegen` 阶段，执行 `genData` 的时候，会执行 `const dirs = genDirectives(el, state)`，它的定义在 `src/compiler/codegen/index.js` 中：

```
function genDirectives (el: ASTElement, state: CodegenState): string | void {
  const dirs = el.directives
  if (!dirs) return
  let res = 'directives:['
  let hasRuntime = false
  let i, l, dir, needRuntime
  for (i = 0, l = dirs.length; i < l; i++) {
    dir = dirs[i]
    needRuntime = true
    const gen: DirectiveFunction = state.directives[dir.name]
    if (gen) {
```

```

    // compile-time directive that manipulates AST.
    // returns true if it also needs a runtime counterpart.
    needRuntime = !!gen(el, dir, state.warn)
}
if (needRuntime) {
  hasRuntime = true
  res += `{"name": "${dir.name}", rawName: "${dir.rawName}" ${
    dir.value ? `, value: ${dir.value}, expression: ${JSON.stringify(dir.value)}`
  }` +
  ` ${
    dir.arg ? `, arg: ${dir.arg}` : ''
  } ${
    dir.modifiers ? `, modifiers: ${JSON.stringify(dir.modifiers)}` : ''
  }},` +
}
if (hasRuntime) {
  return res.slice(0, -1) + ']'
}
}

```

`genDirectives` 方法就是遍历 `el.directives`，然后获取每一个指令对应的方法 `const gen: DirectiveFunction = state.directives[dir.name]`，这个指令方法实际上是在实例化 `CodegenState` 的时候通过 `option` 传入的，这个 `option` 就是编译相关的配置，它在不同的平台上配置不同，在 `web` 环境下的定义在 `src/platforms/web/compiler/options.js` 下：

```

export const baseOptions: CompilerOptions = {
  expectHTML: true,
  modules,
  directives,
  isPreTag,
  isUnaryTag,
  mustUseProp,
  canBeLeftOpenTag,
  isReservedTag,
  getTagNameSpace,
  staticKeys: genStaticKeys(modules)
}

```

`directives` 定义在 `src/platforms/web/compiler/directives/index.js` 中：

```

export default {
  model,
  text,
  html
}

```

那么对于 `v-model` 而言，对应的 `directive` 函数是在 `src/platforms/web/compiler/directives/model.js` 中定义的 `model` 函数：

```
export default function model (
  el: ASTElement,
  dir: ASTDirective,
  _warn: Function
): ?boolean {
  warn = _warn
  const value = dir.value
  const modifiers = dir.modifiers
  const tag = el.tag
  const type = el.attrsMap.type

  if (process.env.NODE_ENV !== 'production') {
    // inputs with type="file" are read only and setting the input's
    // value will throw an error.
    if (tag === 'input' && type === 'file') {
      warn(
        `<${el.tag} v-model="${value}" type="file">:\n` +
        `File inputs are read only. Use a v-on:change listener instead.`
      )
    }
  }

  if (el.component) {
    genComponentModel(el, value, modifiers)
    // component v-model doesn't need extra runtime
    return false
  } else if (tag === 'select') {
    genSelect(el, value, modifiers)
  } else if (tag === 'input' && type === 'checkbox') {
    genCheckboxModel(el, value, modifiers)
  } else if (tag === 'input' && type === 'radio') {
    genRadioModel(el, value, modifiers)
  } else if (tag === 'input' || tag === 'textarea') {
    genDefaultModel(el, value, modifiers)
  } else if (!config.isReservedTag(tag)) {
    genComponentModel(el, value, modifiers)
    // component v-model doesn't need extra runtime
    return false
  } else if (process.env.NODE_ENV !== 'production') {
    warn(
      `<${el.tag} v-model="${value}">: ` +
      `v-model is not supported on this element type. ` +
      `If you are working with contenteditable, it's recommended to ` +
      `wrap a library dedicated for that purpose inside a custom component.`
    )
  }
}
```

```
// ensure runtime directive metadata
return true
}
```

也就是说我们执行 `needRuntime = !!gen(el, dir, state.warn)` 就是在执行 `model` 函数，它会根据 AST 元素节点的不同情况去执行不同的逻辑，对于我们这个 case 而言，它会命中 `genDefaultModel(el, value, modifiers)` 的逻辑，稍后我们也会介绍组件的处理，其它分支同学们可以自行去看。我们来看一下 `genDefaultModel` 的实现：

```
function genDefaultModel (
  el: ASTELEMENT,
  value: string,
  modifiers: ?ASTModifiers
): ?boolean {
  const type = el.attrsMap.type

  // warn if v-bind:value conflicts with v-model
  // except for inputs with v-bind:type
  if (process.env.NODE_ENV !== 'production') {
    const value = el.attrsMap['v-bind:value'] || el.attrsMap[':value']
    const typeBinding = el.attrsMap['v-bind:type'] || el.attrsMap[':type']
    if (value && !typeBinding) {
      const binding = el.attrsMap['v-bind:value'] ? 'v-bind:value' : ':value'
      warn(
        `${binding}="${value}" conflicts with v-model on the same element ` +
        'because the latter already expands to a value binding internally'
      )
    }
  }

  const { lazy, number, trim } = modifiers || {}
  const needCompositionGuard = !lazy && type !== 'range'
  const event = lazy
    ? 'change'
    : type === 'range'
      ? RANGE_TOKEN
      : 'input'

  let valueExpression = '$event.target.value'
  if (trim) {
    valueExpression = `\$event.target.value.trim()`
  }
  if (number) {
    valueExpression = `_\n(${valueExpression})`
  }

  let code = genAssignmentCode(value, valueExpression)
  if (needCompositionGuard) {
    code = `if(\$event.target.composing) return;\${code}`
```

```

    }

    addProp(el, 'value', `(${value})`)
    addHandler(el, event, code, null, true)
    if (trim || number) {
      addHandler(el, 'blur', '$forceUpdate()')
    }
  }
}

```

`genDefaultModel` 函数先处理了 `modifiers`，它的不同主要影响的是 `event` 和 `valueExpression` 的值，对于我们的例子，`event` 为 `input`，`valueExpression` 为 `$event.target.value`。然后去执行 `genAssignmentCode` 去生成代码，它的定义在 `src/compiler/directives/model.js` 中：

```

/**
 * Cross-platform codegen helper for generating v-model value assignment code.
 */
export function genAssignmentCode (
  value: string,
  assignment: string
): string {
  const res = parseModel(value)
  if (res.key === null) {
    return `${value}=${assignment}`
  } else {
    return `$set(${res.exp}, ${res.key}, ${assignment})`
  }
}

```

该方法首先对 `v-model` 对应的 `value` 做了解析，它处理了非常多的情况，对我们的例子，`value` 就是 `message`，所以返回的 `res.key` 为 `null`，然后我们就得到 `${value}=${assignment}`，也就是 `message=$event.target.value`。然后我们又命中了 `needCompositionGuard` 为 `true` 的逻辑，所以最终的 `code` 为

```
if($event.target.composing) return; message=$event.target.value。
```

`code` 生成完后，又执行了 2 句非常关键的代码：

```

addProp(el, 'value', `(${value})`)
addHandler(el, event, code, null, true)

```

这实际上就是 `input` 实现 `v-model` 的精髓，通过修改 AST 元素，给 `el` 添加一个 `prop`，相当于我们在 `input` 上动态绑定了 `value`，又给 `el` 添加了事件处理，相当于在 `input` 上绑定了 `input` 事件，其实转换成模板如下：

```

<input
  v-bind:value="message"
  v-on:input="message=$event.target.value">

```

其实就是在动态绑定了 `input` 的 `value` 指向了 `message` 变量，并且在触发 `input` 事件的时候去动态把 `message` 设置为目标值，这样实际上就完成了数据双向绑定了，所以说 `v-model` 实际上就是语法糖。

再回到 `genDirectives`，它接下来的逻辑就是根据指令生成一些 `data` 的代码：

```
if (needRuntime) {
  hasRuntime = true
  res += `name:"${dir.name}", rawName:"${dir.rawName}" ${
    dir.value ? `, value:(${dir.value}), expression:${JSON.stringify(dir.value)} : ''` :
    `}` ${
      dir.arg ? `, arg:"${dir.arg}"` : ''
    } ${
      dir.modifiers ? `, modifiers:${JSON.stringify(dir.modifiers)}` : ''
    }, `
}

```

对我们的例子而言，最终生成的 `render` 代码如下：

```
with(this) {
  return _c('div', [_c('input', {
    directives:[
      name:"model",
      rawName:"v-model",
      value:(message),
      expression:"message"
    ],
    attrs:{"placeholder":"edit me"},
    domProps:{"value":(message)},
    on:{'input':function($event){
      if($event.target.composing)
        return;
      message=$event.target.value
    }}}, _c('p', [_v("Message is: "+_s(message))])
  ])
}
```

关于事件的处理我们之前的章节已经分析过了，所以对于 `input` 的 `v-model` 而言，完全就是语法糖，并且对于其它表单元素套路都是一样，区别在于生成的事件代码会略有不同。

`v-model` 除了作用在表单元素上，新版的 Vue 还把这一语法糖用在了组件上，接下来我们来分析它的实现。

组件

为了更加直观，我们也是通过一个例子分析：

```
let Child = {
  template: '<div>' +
    '<input :value="value" @input="updateValue" placeholder="edit me">' +
    '</div>',
  props: ['value'],
  methods: {
    updateValue(e) {
      this.$emit('input', e.target.value)
    }
  }
}

let vm = new Vue({
  el: '#app',
  template: '<div>' +
    '<child v-model="message"></child>' +
    '<p>Message is: {{ message }}</p>' +
    '</div>',
  data() {
    return {
      message: ''
    }
  },
  components: {
    Child
  }
})
```

可以看到，父组件引用 `child` 子组件的地方使用了 `v-model` 关联了数据 `message`；而子组件定义了一个 `value` 的 `prop`，并且在 `input` 事件的回调函数中，通过 `this.$emit('input', e.target.value)` 派发了一个事件，为了让 `v-model` 生效，这两点是必须的。

接着我们从源码角度分析实现原理，还是从编译阶段说起，对于父组件而言，在编译阶段会解析 `v-model` 指令，依然会执行 `genData` 函数中的 `genDirectives` 函数，接着执行 `src/platforms/web/compiler/directives/model.js` 中定义的 `model` 函数，并命中如下逻辑：

```
else if (!config.isReservedTag(tag)) {
  genComponentModel(el, value, modifiers);
  return false
}
```

`genComponentModel` 函数定义在 `src/compiler/directives/model.js` 中：

```
export function genComponentModel (
  el: ASTElement,
  value: string,
```

```

    modifiers: ?ASTModifiers
): ?boolean {
  const { number, trim } = modifiers || {}

  const baseValueExpression = '$$v'
  let valueExpression = baseValueExpression
  if (trim) {
    valueExpression =
      `(typeof ${baseValueExpression} === 'string'` +
      `? ${baseValueExpression}.trim()` +
      `: ${baseValueExpression})`
  }
  if (number) {
    valueExpression = `_n(${valueExpression})`
  }
  const assignment = genAssignmentCode(value, valueExpression)

  el.model = {
    value: `(${value})`,
    expression: `"${value}"`,
    callback: `function (${baseValueExpression}) ${assignment}`
  }
}

```

`genComponentModel` 的逻辑很简单，对我们的例子而言，生成的 `el.model` 的值为：

```

el.model = {
  callback: 'function ($$v) {message=$$v}',
  expression: '"message"',
  value: '(message)'
}

```

那么在 `genDirectives` 之后，`genData` 函数中有一段逻辑如下：

```

if (el.model) {
  data += `model:{value:${
    el.model.value
  },callback:${
    el.model.callback
  },expression:${
    el.model.expression
  }}`
}

```

那么父组件最终生成的 `render` 代码如下：

```

with(this){
  return _c('div', [_c('child',{

```

```

model:{
  value:(message),
  callback:function ($$v) {
    message=$$v
  },
  expression:"message"
}
),
_c('p',[_v("Message is: "+_s(message))],1)
}

```

然后在创建子组件 `vnode` 阶段，会执行 `createComponent` 函数，它的定义在 `src/core/vdom/create-component.js` 中：

```

export function createComponent (
  Ctor: Class<Component> | Function | Object | void,
  data: ?VNodeData,
  context: Component,
  children: ?Array<VNode>,
  tag?: string
): VNode | Array<VNode> | void {
  // ...
  // transform component v-model data into props & events
  if (isDef(data.model)) {
    transformModel(Ctor.options, data)
  }

  // extract props
  const propsData = extractPropsFromVNodeData(data, Ctor, tag)
  // ...
  // extract listeners, since these needs to be treated as
  // child component listeners instead of DOM listeners
  const listeners = data.on
  // ...
  const vnode = new VNode(
    `vue-component-${Ctor.cid}${name ? `-${name}` : ''}`,
    data, undefined, undefined, context,
    { Ctor, propsData, listeners, tag, children },
    asyncFactory
  )

  return vnode
}

```

其中会对 `data.model` 的情况做处理，执行 `transformModel(Ctor.options, data)` 方法：

```

// transform component v-model info (value and callback) into
// prop and event handler respectively.
function transformModel (options, data: any) {

```

```

const prop = (options.model && options.model.prop) || 'value'
const event = (options.model && options.model.event) || 'input'
;(data.props || (data.props = {}))[prop] = data.model.value
const on = data.on || (data.on = {})
if (isDef(on[event])) {
  on[event] = [data.model.callback].concat(on[event])
} else {
  on[event] = data.model.callback
}
}

```

`transformModel` 逻辑很简单，给 `data.props` 添加 `data.model.value`，并且给 `data.on` 添加 `data.model.callback`，对我们的例子而言，扩展结果如下：

```

data.props = {
  value: (message),
}
data.on = {
  input: function ($$v) {
    message=$$v
  }
}

```

其实就相当于我们这样编写父组件：

```

let vm = new Vue({
  el: '#app',
  template: '<div>' +
  '<child :value="message" @input="message=arguments[0]"></child>' +
  '<p>Message is: {{ message }}</p>' +
  '</div>',
  data() {
    return {
      message: ''
    }
  },
  components: {
    Child
  }
})

```

子组件传递的 `value` 绑定到当前父组件的 `message`，同时监听自定义 `input` 事件，当子组件派发 `input` 事件的时候，父组件会在事件回调函数中修改 `message` 的值，同时 `value` 也会发生变化，子组件的 `input` 值被更新。

这就是典型的 Vue 的父子组件通讯模式，父组件通过 `prop` 把数据传递到子组件，子组件修改了数据后把改变通过 `$emit` 事件的方式通知父组件，所以说组件上的 `v-model` 也是一种语法糖。

另外我们注意到组件 `v-model` 的实现，子组件的 `value prop` 以及派发的 `input` 事件名是可配的，可以看到 `transformModel` 中对这部分的处理：

```
function transformModel (options, data: any) {
  const prop = (options.model && options.model.prop) || 'value'
  const event = (options.model && options.model.event) || 'input'
  // ...
}
```

也就是说可以在定义子组件的时候通过 `model` 选项配置子组件接收的 `prop` 名以及派发的事件名，举个例子：

```
let Child = {
  template: '<div>' +
    '<input :value="msg" @input="updateValue" placeholder="edit me">' +
    '</div>',
  props: ['msg'],
  model: {
    prop: 'msg',
    event: 'change'
  },
  methods: {
    updateValue(e) {
      this.$emit('change', e.target.value)
    }
  }
}

let vm = new Vue({
  el: '#app',
  template: '<div>' +
    '<child v-model="message"></child>' +
    '<p>Message is: {{ message }}</p>' +
    '</div>',
  data() {
    return {
      message: ''
    }
  },
  components: {
    Child
  }
})
```

子组件修改了接收的 `prop` 名以及派发的事件名，然而这一切父组件作为调用方是不用关心的，这样做的好处是我们可以把 `value` 这个 `prop` 作为其它的用途。

总结

那么至此，`v-model` 的实现就分析完了，我们了解到它是 Vue 双向绑定的真正实现，但本质上就是一种语法糖，它即可以支持原生表单元素，也可以支持自定义组件。在组件的实现中，我们是可以配置子组件接收的 `prop` 名称，以及派发的事件名称。

slot

Vue 的组件提供了一个非常有用的特性—— `slot` 插槽，它让组件的实现变的更加灵活。我们平时在开发组件库的时候，为了让组件更加灵活可定制，经常用插槽的方式让用户可以自定义内容。插槽分为普通插槽和作用域插槽，它们可以解决不同的场景，但它是怎么实现的呢，下面我们就从源码的角度来分析插槽的实现原理。

普通插槽

为了更加直观，我们还是通过一个例子来分析插槽的实现：

```
let AppLayout = {
  template: '<div class="container">' +
    '<header><slot name="header"></slot></header>' +
    '<main><slot>默认内容</slot></main>' +
    '<footer><slot name="footer"></slot></footer>' +
    '</div>'
}

let vm = new Vue({
  el: '#app',
  template: '<div>' +
    '<app-layout>' +
    '<h1 slot="header">{{title}}</h1>' +
    '<p>{{msg}}</p>' +
    '<p slot="footer">{{desc}}</p>' +
    '</app-layout>' +
    '</div>',
  data() {
    return {
      title: '我是标题',
      msg: '我是内容',
      desc: '其它信息'
    }
  },
  components: {
    AppLayout
  }
})
```

这里我们定义了 `AppLayout` 子组件，它内部定义了 3 个插槽，2 个为具名插槽，一个 `name` 为 `header`，一个 `name` 为 `footer`，还有一个没有定义 `name` 的是默认插槽。`<slot>` 和 `</slot>` 之前填写的内容为默认内容。我们的父组件注册和引用了 `AppLayout` 的组件，并在组件内部定义了一些元素，用来替换插槽，那么它最终生成的 DOM 如下：

```
<div>
```

```

<div class="container">
  <header><h1>我是标题</h1></header>
  <main><p>我是内容</p></main>
  <footer><p>其它信息</p></footer>
</div>
</div>

```

编译

还是先从编译说起，我们知道编译是发生在调用 `vm.$mount` 的时候，所以编译的顺序是先编译父组件，再编译子组件。

首先编译父组件，在 `parse` 阶段，会执行 `processSlot` 处理 `slot`，它的定义在 `src/compiler/parser/index.js` 中：

```

function processSlot (el) {
  if (el.tag === 'slot') {
    el.slotName = getBindingAttr(el, 'name')
    if (process.env.NODE_ENV !== 'production' && el.key) {
      warn(
        `\\`key\\` does not work on <slot> because slots are abstract outlets ` +
        `and can possibly expand into multiple elements. ` +
        `Use the key on a wrapping element instead.`
      )
    }
  } else {
    let slotScope
    if (el.tag === 'template') {
      slotScope = getAndRemoveAttr(el, 'scope')
      /* istanbul ignore if */
      if (process.env.NODE_ENV !== 'production' && slotScope) {
        warn(
          `the "scope" attribute for scoped slots have been deprecated and ` +
          `replaced by "slot-scope" since 2.5. The new "slot-scope" attribute ` +
          `can also be used on plain elements in addition to <template> to ` +
          `denote scoped slots.`,
          true
        )
      }
      el.slotScope = slotScope || getAndRemoveAttr(el, 'slot-scope')
    } else if ((slotScope = getAndRemoveAttr(el, 'slot-scope'))) {
      /* istanbul ignore if */
      if (process.env.NODE_ENV !== 'production' && el.attrsMap['v-for']) {
        warn(
          `Ambiguous combined usage of slot-scope and v-for on <${el.tag}> ` +
          `(v-for takes higher priority). Use a wrapper <template> for the ` +
          `scoped slot to make it clearer.`,
          true
        )
      }
    }
  }
}

```

```

        }
        el.slotScope = slotScope
    }
    const slotTarget = getBindingAttr(el, 'slot')
    if (slotTarget) {
        el.slotTarget = slotTarget === '' ? '"default"' : slotTarget
        // preserve slot as an attribute for native shadow DOM compat
        // only for non-scoped slots.
        if (el.tag !== 'template' && !el.slotScope) {
            addAttr(el, 'slot', slotTarget)
        }
    }
}

```

当解析到标签上有 `slot` 属性的时候，会给对应的 AST 元素节点添加 `slotTarget` 属性，然后在 `codegen` 阶段，在 `genData` 中会处理 `slotTarget`，相关代码在 `src/compiler/codegen/index.js` 中：

```

if (el.slotTarget && !el.slotScope) {
    data += `slot:${el.slotTarget},`
}

```

会给 `data` 添加一个 `slot` 属性，并指向 `slotTarget`，之后会用到。在我们的例子中，父组件最终生成的代码如下：

```

with(this){
    return _c('div',
        [_c('app-layout',
            [_c('h1',{attrs:{'slot':'header'}},slot:"header"),
                [_v(_s(title))]),
            _c('p',[_v(_s(msg))]),
            _c('p',{attrs:{'slot':'footer'}},slot:"footer"),
                [_v(_s(desc))]
            )
        ],
        1)}

```

接下来编译子组件，同样在 `parser` 阶段会执行 `processSlot` 处理函数，它的定义在 `src/compiler/parser/index.js` 中：

```

function processSlot (el) {
    if (el.tag === 'slot') {
        el.slotName = getBindingAttr(el, 'name')
    }
    // ...
}

```

当遇到 `slot` 标签的时候会给对应的 AST 元素节点添加 `slotName` 属性，然后在 `codegen` 阶段，会判断如果当前 AST 元素节点是 `slot` 标签，则执行 `genSlot` 函数，它的定义在 `src/compiler/codegen/index.js` 中：

```
function genSlot (el: ASTElement, state: CodegenState): string {
  const slotName = el.slotName || '"default"'
  const children = genChildren(el, state)
  let res = `_t(${slotName}${children ? `,${children}` : ''}`
  const attrs = el.attrs && `#${el.attrs.map(a => `${camelize(a.name)}:${a.value}`)}
  .join(',')}`
  const bind = el.attrsMap['v-bind']
  if ((attrs || bind) && !children) {
    res += `,null`
  }
  if (attrs) {
    res += `,${attrs}`
  }
  if (bind) {
    res += `${attrs ? '' : ',null'},${bind}`
  }
  return res + ')'
}
```

我们先不考虑 `slot` 标签上有 `attrs` 以及 `v-bind` 的情况，那么它生成的代码实际上就只有：

```
const slotName = el.slotName || '"default"'
const children = genChildren(el, state)
let res = `_t(${slotName}${children ? `,${children}` : ''}`
```

这里的 `slotName` 从 AST 元素节点对应的属性上取，默认是 `default`，而 `children` 对应的就是 `slot` 开始和闭合标签包裹的内容。来看一下我们例子的子组件最终生成的代码，如下：

```
with(this) {
  return _c('div',{
    staticClass:"container"
  ),[
    _c('header',[_t("header")],2),
    _c('main',[_t("default"),[_v("默认内容")]]],2),
    _c('footer',[_t("footer")],2)
  ]
}
```

在编译章节我们了解到，`_t` 函数对应的就是 `renderSlot` 方法，它的定义在 `src/core/instance/render-heplpers/render-slot.js` 中：

```
/**  
 * Runtime helper for rendering <slot>  
 */  
export function renderSlot (  
  name: string,  
  fallback: ?Array<VNode>,  
  props: ?Object,  
  bindObject: ?Object  
) : ?Array<VNode> {  
  const scopedSlotFn = this.$scopedSlots[name]  
  let nodes  
  if (scopedSlotFn) { // scoped slot  
    props = props || {}  
    if (bindObject) {  
      if (process.env.NODE_ENV !== 'production' && !isObject(bindObject)) {  
        warn(  
          'slot v-bind without argument expects an Object',  
          this  
        )  
      }  
      props = extend(extend({}, bindObject), props)  
    }  
    nodes = scopedSlotFn(props) || fallback  
  } else {  
    const slotNodes = this.$slots[name]  
    // warn duplicate slot usage  
    if (slotNodes) {  
      if (process.env.NODE_ENV !== 'production' && slotNodes._rendered) {  
        warn(  
          `Duplicate presence of slot "${name}" found in the same render tree ` +  
          `- this will likely cause render errors.`,
          this  
        )  
      }  
      slotNodes._rendered = true  
    }  
    nodes = slotNodes || fallback  
  }  
  
  const target = props && props.slot  
  if (target) {  
    return this.$createElement('template', { slot: target }, nodes)  
  } else {  
    return nodes  
  }  
}
```

`render-slot` 的参数 `name` 代表插槽名称 `slotName`，`fallback` 代表插槽的默认内容生成的 `vnode` 数组。先忽略 `scoped-slot`，只看默认插槽逻辑。如果 `this.$slot[name]` 有值，就返回它对应的 `vnode` 数组，否则返回 `fallback`。那么这个 `this.$slot` 是哪里来的呢？我们知道子组件的 `init` 时机是在父组件执行 `patch` 过程的时候，那这个时候父组件已经编译完成了。并且子组件在 `init` 过程中会执行 `initRender` 函数，`initRender` 的时候获取到 `vm.$slot`，相关代码在 `src/core/instance/render.js` 中：

```
export function initRender (vm: Component) {
  // ...
  const parentVnode = vm.$vnode = options._parentVnode // the placeholder node in parent tree
  const renderContext = parentVnode && parentVnode.context
  vm.$slots = resolveSlots(options._renderChildren, renderContext)
}
```

`vm.$slots` 是通过执行 `resolveSlots(options._renderChildren, renderContext)` 返回的，它的定义在 `src/core/instance/render-helpers/resolve-slots.js` 中：

```
/**
 * Runtime helper for resolving raw children VNodes into a slot object.
 */
export function resolveSlots (
  children: ?Array<VNode>,
  context: ?Component
): { [key: string]: Array<VNode> } {
  const slots = {}
  if (!children) {
    return slots
  }
  for (let i = 0, l = children.length; i < l; i++) {
    const child = children[i]
    const data = child.data
    // remove slot attribute if the node is resolved as a Vue slot node
    if (data && data.attrs && data.attrs.slot) {
      delete data.attrs.slot
    }
    // named slots should only be respected if the vnode was rendered in the
    // same context.
    if ((child.context === context || child.fnContext === context) &&
      data && data.slot !== null
    ) {
      const name = data.slot
      const slot = (slots[name] || (slots[name] = []))
      if (child.tag === 'template') {
        slot.push.apply(slot, child.children || [])
      } else {
        slot.push(child)
      }
    } else {
  }
```

```

        (slots.default || (slots.default = [])).push(child)
    }
}
// ignore slots that contains only whitespace
for (const name in slots) {
    if (slots[name].every(isWhitespace)) {
        delete slots[name]
    }
}
return slots
}

```

`resolveSlots` 方法接收 2 个参数，第一个参数 `children` 对应的是父 `vnode` 的 `children`，在我们的例子中就是 `<app-layout>` 和 `</app-layout>` 包裹的内容。第二个参数 `context` 是父 `vnode` 的上下文，也就是父组件的 `vm` 实例。

`resolveSlots` 函数的逻辑就是遍历 `children`，拿到每一个 `child` 的 `data`，然后通过 `data.slot` 获取到插槽名称，这个 `slot` 就是我们之前编译父组件在 `codegen` 阶段设置的 `data.slot`。接着以插槽名称为 `key` 把 `child` 添加到 `slots` 中，如果 `data.slot` 不存在，则是默认插槽的内容，则把对应的 `child` 添加到 `slots.defaults` 中。这样就获取到整个 `slots`，它是一个对象，`key` 是插槽名称，`value` 是一个 `vnode` 类型的数组，因为它可以有多个同名插槽。

这样我们就拿到了 `vm.$slots` 了，回到 `renderSlot` 函数，`const slotNodes = this.$slots[name]`，我们也就能够根据插槽名称获取到对应的 `vnode` 数组了，这个数组里的 `vnode` 都是在父组件创建的，这样就实现了在父组替换子组件插槽的内容了。

对应的 `slot` 渲染成 `vnodes`，作为当前组件渲染 `vnode` 的 `children`，之后的渲染过程之前分析过，不再赘述。

我们知道在普通插槽中，父组件应用到子组件插槽里的数据都是绑定到父组件的，因为它渲染成 `vnode` 的时机的上下文是父组件的实例。但是在一些实际开发中，我们想通过子组件的一些数据来决定父组件实现插槽的逻辑，Vue 提供了另一种插槽——作用域插槽，接下来我们就来分析一下它的实现原理。

作用域插槽

为了更加直观，我们也是通过一个例子来分析作用域插槽的实现：

```

let Child = {
  template: '<div class="child">' +
    '<slot text="Hello " :msg="msg"></slot>' +
    '</div>',
  data() {
    return {
      msg: 'Vue'
    }
  }
}

```

```

}

let vm = new Vue({
  el: '#app',
  template: '<div>' +
  '<child>' +
  '<template slot-scope="props">' +
  '<p>Hello from parent</p>' +
  '<p>{{ props.text + props.msg }}</p>' +
  '</template>' +
  '</child>' +
  '</div>',
  components: {
    Child
  }
})

```

最终生成的 DOM 结构如下：

```

<div>
  <div class="child">
    <p>Hello from parent</p>
    <p>Hello Vue</p>
  </div>
</div>

```

我们可以看到子组件的 `slot` 标签多了 `text` 属性，以及 `:msg` 属性。父组件实现插槽的部分多了一个 `template` 标签，以及 `scope-slot` 属性，其实在 Vue 2.5+ 版本，`scoped-slot` 可以作用在普通元素上。这些就是作用域插槽和普通插槽在写法上的差别。

在编译阶段，仍然是先编译父组件，同样是通过 `processSlot` 函数去处理 `scoped-slot`，它的定义在 `src/compiler/parser/index.js` 中：

```

function processSlot (el) {
  // ...
  let slotScope
  if (el.tag === 'template') {
    slotScope = getAndRemoveAttr(el, 'scope')
    /* istanbul ignore if */
    if (process.env.NODE_ENV !== 'production' && slotScope) {
      warn(
        `the "scope" attribute for scoped slots have been deprecated and ` +
        `replaced by "slot-scope" since 2.5. The new "slot-scope" attribute ` +
        `can also be used on plain elements in addition to <template> to ` +
        `denote scoped slots.`,
        true
      )
    }
    el.slotScope = slotScope || getAndRemoveAttr(el, 'slot-scope')
  }
}

```

```

} else if ((slotScope = getAndRemoveAttr(el, 'slot-scope'))) {
  /* istanbul ignore if */
  if (process.env.NODE_ENV !== 'production' && el.attrsMap['v-for']) {
    warn(
      `Ambiguous combined usage of slot-scope and v-for on <${el.tag}> ` +
      `(v-for takes higher priority). Use a wrapper <template> for the ` +
      `scoped slot to make it clearer.`,
      true
    )
  }
  el.slotScope = slotScope
}
// ...
}

```

这块逻辑很简单，读取 `scoped-slot` 属性并赋值给当前 AST 元素节点的 `slotScope` 属性，接下来在构造 AST 树的时候，会执行以下逻辑：

```

if (element.elseif || element.else) {
  processIfConditions(element, currentParent)
} else if (element.slotScope) {
  currentParent.plain = false
  const name = element.slotTarget || '"default"'
  ;(currentParent.scopedSlots || (currentParent.scopedSlots = {}))[name] = element
} else {
  currentParent.children.push(element)
  element.parent = currentParent
}

```

可以看到对于拥有 `scopedSlot` 属性的 AST 元素节点而言，是不会作为 `children` 添加到当前 AST 树中，而是存到父 AST 元素节点的 `scopedSlots` 属性上，它是一个对象，以插槽名称 `name` 为 `key`。

然后在 `genData` 的过程，会对 `scopedSlots` 做处理：

```

if (el.scopedSlots) {
  data += `${genScopedSlots(el.scopedSlots, state)},` 
}

function genScopedSlots (
  slots: { [key: string]: ASTElement },
  state: CodegenState
): string {
  return `scopedSlots:_u([${ 
    Object.keys(slots).map(key => {
      return genScopedSlot(key, slots[key], state)
    }).join(',')
  }])` 
}

```

```

function genScopedSlot (
  key: string,
  el: ASTElement,
  state: CodegenState
): string {
  if (el.for && !el.forProcessed) {
    return genForScopedSlot(key, el, state)
  }
  const fn = `function(${String(el.slotScope)}){` +
    `return ${el.tag === 'template'}` +
    `? el.if` +
    `? ${el.if}?${genChildren(el, state) || 'undefined'}:undefined` +
    `: genChildren(el, state) || 'undefined'` +
    `: genElement(el, state)` +
  `}` +
  `return ${key:${key},fn:${fn}}` +
}

```

`genScopedSlots` 就是对 `scopedSlots` 对象遍历，执行 `genScopedSlot`，并把结果用逗号拼接，而 `genScopedSlot` 是先生成一段函数代码，并且函数的参数就是我们的 `slotScope`，也就是写在标签属性上的 `scoped-slot` 对应的值，然后再返回一个对象，`key` 为插槽名称，`fn` 为生成的函数代码。

对于我们这个例子而言，父组件最终生成的代码如下：

```

with(this){
  return _c('div',
    [_c('child',
      {scopedSlots:_u([
        {
          key: "default",
          fn: function(props) {
            return [
              _c('p',[_v("Hello from parent")]),
              _c('p',[_v(_s(props.text + props.msg))])
            ]
          }
        }]))
      ]),
    1)
}

```

可以看到它和普通插槽父组件编译结果的一个很明显的区别就是没有 `children` 了，`data` 部分多了一个对象，并且执行了 `_u` 方法，在编译章节我们了解到，`_u` 函数对的就是 `resolveScopedSlots` 方法，它的定义在 `src/core/instance/render-heplpers/resolve-slots.js` 中：

```

export function resolveScopedSlots (
  fns: ScopedSlotsData, // see flow/vnode
  res?: Object
): { [key: string]: Function } {
  res = res || {}
  for (let i = 0; i < fns.length; i++) {
    if (Array.isArray(fns[i])) {
      resolveScopedSlots(fns[i], res)
    } else {
      res[fns[i].key] = fns[i].fn
    }
  }
  return res
}

```

其中，`fns` 是一个数组，每一个数组元素都有一个 `key` 和一个 `fn`，`key` 对应的是插槽的名称，`fn` 对应一个函数。整个逻辑就是遍历这个 `fns` 数组，生成一个对象，对象的 `key` 就是插槽名称，`value` 就是函数。这个函数的执行时机稍后我们会介绍。

接着我们再来看一下子组件的编译，和普通插槽的过程基本相同，唯一一点区别是在 `genSlot` 的时候：

```

function genSlot (el: ASTElement, state: CodegenState): string {
  const slotName = el.slotName || '"default"'
  const children = genChildren(el, state)
  let res = `_t(${slotName})${children ? `,${children}` : ''}`
  const attrs = el.attrs && `${el.attrs.map(a => `${camelize(a.name)}:${a.value}`).join(',')}`
  const bind = el.attrsMap['v-bind']
  if ((attrs || bind) && !children) {
    res += `,null`
  }
  if (attrs) {
    res += `,${attrs}`
  }
  if (bind) {
    res += `${attrs ? '' : ',null'},${bind}`
  }
  return res + ')'
}

```

它会对 `attrs` 和 `v-bind` 做处理，对应到我们的例子，最终生成的代码如下：

```

with(this){
  return _c('div',
    {staticClass:"child"},
    [_t("default",null,
      {text:"Hello ",msg:msg}
    )],
  )
}

```

2)}

`_t` 方法我们之前介绍过，对应的是 `renderSlot` 方法：

```
export function renderSlot (
  name: string,
  fallback: ?Array<VNode>,
  props: ?Object,
  bindObject: ?Object
): ?Array<VNode> {
  const scopedSlotFn = this.$scopedSlots[name]
  let nodes
  if (scopedSlotFn) {
    props = props || {}
    if (bindObject) {
      if (process.env.NODE_ENV !== 'production' && !isObject(bindObject)) {
        warn(
          'slot v-bind without argument expects an Object',
          this
        )
      }
    }
    props = extend(extend({}, bindObject), props)
  }
  nodes = scopedSlotFn(props) || fallback
} else {
  // ...
}

const target = props && props.slot
if (target) {
  return this.$createElement('template', { slot: target }, nodes)
} else {
  return nodes
}
}
```

我们只关注作用域插槽的逻辑，那么这个 `this.$scopedSlots` 又是在什么地方定义的呢，原来在子组件的渲染函数执行前，在 `vm_render` 方法内，有这么一段逻辑，定义在 `src/core/instance/render.js` 中：

```
if (_parentVnode) {
  vm.$scopedSlots = _parentVnode.data.scopedSlots || emptyObject
}
```

这个 `_parentVnode.data.scopedSlots` 对应的就是我们在父组件通过执行 `resolveScopedSlots` 返回的对象。所以回到 `genSlot` 函数，我们就可以通过插槽的名称拿到对应的 `scopedSlotFn`，然后把相关的数据扩展到 `props` 上，作为函数的参数传入，原来之前我们提到的函数这个时候执行，然后返回生成的 `vnodes`，为后续渲染节点用。

后续流程之前已介绍过，不再赘述，那么至此，作用域插槽的实现也就分析完毕。

总结

通过这一章的分析，我们了解了普通插槽和作用域插槽的实现。它们有一个很大的差别是数据作用域，普通插槽是在父组件编译和渲染阶段生成 `vnodes`，所以数据的作用域是父组件实例，子组件渲染的时候直接拿到这些渲染好的 `vnodes`。而对于作用域插槽，父组件在编译和渲染阶段并不会直接生成 `vnodes`，而是在父节点 `vnode` 的 `data` 中保留一个 `scopedSlots` 对象，存储着不同名称的插槽以及它们对应的渲染函数，只有在编译和渲染子组件阶段才会执行这个渲染函数生成 `vnodes`，由于是在子组件环境执行的，所以对应的数据作用域是子组件实例。

简单地说，两种插槽的目的都是让子组件 `slot` 占位符生成的内容由父组件来决定，但数据的作用域会根据它们 `vnodes` 渲染时机不同而不同。

keep-alive

在我们的平时开发工作中，经常为了组件的缓存优化而使用 `<keep-alive>` 组件，乐此不疲，但很少有人关注它的实现原理，下面就让我们来一探究竟。

内置组件

`<keep-alive>` 是 Vue 源码中实现的一个组件，也就是说 Vue 源码不仅实现了一套组件化的机制，也实现了一些内置组件，它的定义在 `src/core/components/keep-alive.js` 中：

```
export default {
  name: 'keep-alive',
  abstract: true,

  props: {
    include: patternTypes,
    exclude: patternTypes,
    max: [String, Number]
  },

  created () {
    this.cache = Object.create(null)
    this.keys = []
  },

  destroyed () {
    for (const key in this.cache) {
      pruneCacheEntry(this.cache, key, this.keys)
    }
  },

  mounted () {
    this.$watch('include', val => {
      pruneCache(this, name => matches(val, name))
    })
    this.$watch('exclude', val => {
      pruneCache(this, name => !matches(val, name))
    })
  },

  render () {
    const slot = this.$slots.default
    const vnode: VNode = getFirstComponentChild(slot)
    const componentOptions: ?VNodeComponentOptions = vnode && vnode.componentOptions
    if (componentOptions) {
      // check pattern
    }
  }
}
```

```

const name: ?string = getComponentName(componentOptions)
const { include, exclude } = this
if (
  // not included
  (include && (!name || !matches(include, name))) ||
  // excluded
  (exclude && name && matches(exclude, name)))
) {
  return vnode
}

const { cache, keys } = this
const key: ?string = vnode.key == null
  // same constructor may get registered as different local components
  // so cid alone is not enough (#3269)
  ? componentOptions.Ctor.cid + (componentOptions.tag ? `::${componentOptions
.tag}` : '')
  : vnode.key
if (cache[key]) {
  vnode.componentInstance = cache[key].componentInstance
  // make current key freshest
  remove(keys, key)
  keys.push(key)
} else {
  cache[key] = vnode
  keys.push(key)
  // prune oldest entry
  if (this.max && keys.length > parseInt(this.max)) {
    pruneCacheEntry(cache, keys[0], keys, this._vnode)
  }
}
vnode.data.keepAlive = true
}
return vnode || (slot && slot[0])
}
}

```

可以看到 `<keep-alive>` 组件的实现也是一个对象，注意它有一个属性 `abstract` 为 `true`，是一个抽象组件，Vue 的文档没有提这个概念，实际上它在组件实例建立父子关系的时候会被忽略，发生在 `initLifecycle` 的过程中：

```

// locate first non-abstract parent
let parent = options.parent
if (parent && !options.abstract) {
  while (parent.$options.abstract && parent.$parent) {
    parent = parent.$parent
  }
  parent.$children.push(vm)
}

```

```
vm.$parent = parent
```

`<keep-alive>` 在 `created` 钩子里定义了 `this.cache` 和 `this.keys`，本质上它就是去缓存已经创建过的 `vnode`。它的 `props` 定义了 `include`，`exclude`，它们可以字符串或者表达式，`include` 表示只有匹配的组件会被缓存，而 `exclude` 表示任何匹配的组件都不会被缓存，`props` 还定义了 `max`，它表示缓存的大小，因为我们是缓存的 `vnode` 对象，它也会持有 DOM，当我们缓存很多的时候，会比较占用内存，所以该配置允许我们指定缓存大小。

`<keep-alive>` 直接实现了 `render` 函数，而不是我们常规模板的方式，执行 `<keep-alive>` 组件渲染的时候，就会执行到这个 `render` 函数，接下来我们分析一下它的实现。

首先获取第一个子元素的 `vnode`：

```
const slot = this.$slots.default
const vnode: VNode = getFirstComponentChild(slot)
```

由于我们也是在 `<keep-alive>` 标签内部写 DOM，所以可以先获取到它的默认插槽，然后再获取到它的第一个子节点。`<keep-alive>` 只处理第一个子元素，所以一般和它搭配使用的有 `component` 动态组件或者是 `router-view`，这点要牢记。

然后又判断了当前组件的名称和 `include`、`exclude` 的关系：

```
// check pattern
const name: ?string = getComponentName(componentOptions)
const { include, exclude } = this
if (
  // not included
  (include && (!name || !matches(include, name))) ||
  // excluded
  (exclude && name && matches(exclude, name)))
) {
  return vnode
}

function matches (pattern: string | RegExp | Array<string>, name: string): boolean
{
  if (Array.isArray(pattern)) {
    return pattern.indexOf(name) > -1
  } else if (typeof pattern === 'string') {
    return pattern.split(',').indexOf(name) > -1
  } else if (isRegExp(pattern)) {
    return pattern.test(name)
  }
  return false
}
```

`matches` 的逻辑很简单，就是做匹配，分别处理了数组、字符串、正则表达式的情况，也就是说我们平时传的 `include` 和 `exclude` 可以是这三种类型的任意一种。并且我们的组件名如果满足了配置 `include` 且不匹配或者是配置了 `exclude` 且匹配，那么就直接返回这个组件的 `vnode`，否则的话走下一步缓存：

```
const { cache, keys } = this
const key: ?string = vnode.key == null
// same constructor may get registered as different local components
// so cid alone is not enough (#3269)
? componentOptions.Ctor.cid + (componentOptions.tag ? `::${componentOptions.tag}` :
: '')
: vnode.key
if (cache[key]) {
  vnode.componentInstance = cache[key].componentInstance
  // make current key freshest
  remove(keys, key)
  keys.push(key)
} else {
  cache[key] = vnode
  keys.push(key)
  // prune oldest entry
  if (this.max && keys.length > parseInt(this.max)) {
    pruneCacheEntry(cache, keys[0], keys, this._vnode)
  }
}
```

这部分逻辑很简单，如果命中缓存，则直接从缓存中拿 `vnode` 的组件实例，并且重新调整了 `key` 的顺序放在了最后一个；否则把 `vnode` 设置进缓存，最后还有一个逻辑，如果配置了 `max` 并且缓存的长度超过了 `this.max`，还要从缓存中删除第一个：

```
function pruneCacheEntry (
  cache: VNodeCache,
  key: string,
  keys: Array<string>,
  current?: VNode
) {
  const cached = cache[key]
  if (cached && (!current || cached.tag !== current.tag)) {
    cached.componentInstance.$destroy()
  }
  cache[key] = null
  remove(keys, key)
}
```

除了从缓存中删除外，还要判断如果要删除的缓存并的组件 `tag` 不是当前渲染组件 `tag`，也执行删除缓存的组件实例的 `$destroy` 方法。

最后设置 `vnode.data.keepAlive = true`，这个作用稍后我们介绍。

注意，`<keep-alive>` 组件也是为观测 `include` 和 `exclude` 的变化，对缓存做处理：

```

watch: {
  include (val: string | RegExp | Array<string>) {
    pruneCache(this, name => matches(val, name))
  },
  exclude (val: string | RegExp | Array<string>) {
    pruneCache(this, name => !matches(val, name))
  }
}

function pruneCache (keepAliveInstance: any, filter: Function) {
  const { cache, keys, _vnode } = keepAliveInstance
  for (const key in cache) {
    const cachedNode: ?VNode = cache[key]
    if (cachedNode) {
      const name: ?string = getComponentName(cachedNode.componentOptions)
      if (name && !filter(name)) {
        pruneCacheEntry(cache, key, keys, _vnode)
      }
    }
  }
}

```

逻辑很简单，观测他们的变化执行 `pruneCache` 函数，其实就是对 `cache` 做遍历，发现缓存的节点名称和新的规则没有匹配上的时候，就把这个缓存节点从缓存中摘除。

组件渲染

到此为止，我们只了解了 `<keep-alive>` 的组件实现，但并不知道它包裹的子组件渲染和普通组件有什么不一样的地方。我们关注 2 个方面，首次渲染和缓存渲染。

同样为了更好地理解，我们也结合一个示例来分析：

```

let A = {
  template: '<div class="a">' +
  '<p>A Comp</p>' +
  '</div>',
  name: 'A'
}

let B = {
  template: '<div class="b">' +
  '<p>B Comp</p>' +
  '</div>',
  name: 'B'
}

let vm = new Vue({

```

```

el: '#app',
template: '<div>' +
'<keep-alive>' +
'<component :is="currentComp">' +
'</component>' +
'</keep-alive>' +
'<button @click="change">switch</button>' +
'</div>',
data: {
  currentComp: 'A'
},
methods: {
  change() {
    this.currentComp = this.currentComp === 'A' ? 'B' : 'A'
  }
},
components: {
  A,
  B
}
))

```

首次渲染

我们知道 Vue 的渲染最后都会到 `patch` 过程，而组件的 `patch` 过程会执行 `createComponent` 方法，它的定义在 `src/core/vdom/patch.js` 中：

```

function createComponent (vnode, insertedVnodeQueue, parentElm, refElm) {
  let i = vnode.data
  if (isDef(i)) {
    const isReactivated = isDef(vnode.componentInstance) && i.keepAlive
    if (isDef(i = i.hook) && isDef(i = i.init)) {
      i(vnode, false /* hydrating */)
    }
    // after calling the init hook, if the vnode is a child component
    // it should've created a child instance and mounted it. the child
    // component also has set the placeholder vnode's elm.
    // in that case we can just return the element and be done.
    if (isDef(vnode.componentInstance)) {
      initComponent(vnode, insertedVnodeQueue)
      insert(parentElm, vnode.elm, refElm)
      if (isTrue(isReactivated)) {
        reactivateComponent(vnode, insertedVnodeQueue, parentElm, refElm)
      }
      return true
    }
  }
}

```

`createComponent` 定义了 `isReactivated` 的变量，它是根据 `vnode.componentInstance` 以及 `vnode.data.keepAlive` 的判断，第一次渲染的时候，`vnode.componentInstance` 为 `undefined`，`vnode.data.keepAlive` 为 `true`，因为它的父组件 `<keep-alive>` 的 `render` 函数会先执行，那么该 `vnode` 缓存到内存中，并且设置 `vnode.data.keepAlive` 为 `true`，因此 `isReactivated` 为 `false`，那么走正常的 `init` 的钩子函数执行组件的 `mount`。当 `vnode` 已经执行完 `patch` 后，执行 `initComponent` 函数：

```
function initComponent (vnode, insertedVnodeQueue) {
  if (isDef(vnode.data.pendingInsert)) {
    insertedVnodeQueue.push.apply(insertedVnodeQueue, vnode.data.pendingInsert)
    vnode.data.pendingInsert = null
  }
  vnode.elm = vnode.componentInstance.$el
  if (isPatchable(vnode)) {
    invokeCreateHooks(vnode, insertedVnodeQueue)
    setScope(vnode)
  } else {
    // empty component root.
    // skip all element-related modules except for ref (#3455)
    registerRef(vnode)
    // make sure to invoke the insert hook
    insertedVnodeQueue.push(vnode)
  }
}
```

这里会有 `vnode.elm` 缓存了 `vnode` 创建生成的 DOM 节点。所以对于首次渲染而言，除了在 `<keep-alive>` 中建立缓存，和普通组件渲染没什么区别。

所以对我们的例子，初始化渲染 `A` 组件以及第一次点击 `switch` 渲染 `B` 组件，都是首次渲染。

缓存渲染

当我们从 `B` 组件再次点击 `switch` 切换到 `A` 组件，就会命中缓存渲染。

我们之前分析过，当数据发送变化，在 `patch` 的过程中会执行 `patchVnode` 的逻辑，它会对比新旧 `vnode` 节点，甚至对比它们的子节点去做更新逻辑，但是对于组件 `vnode` 而言，是没有 `children` 的，那么对于 `<keep-alive>` 组件而言，如何更新它包裹的内容呢？

原来 `patchVnode` 在做各种 diff 之前，会先执行 `prepatch` 的钩子函数，它的定义在 `src/core/vdom/create-component` 中：

```
const componentVNodeHooks = {
  prepatch (oldVnode: MountedComponentVNode, vnode: MountedComponentVNode) {
    const options = vnode.componentInstance.options
    const child = vnode.componentInstance = oldVnode.componentInstance
    updateChildComponent(
      child,
      options.propsData, // updated props
    )
  }
}
```

```

        options.listeners, // updated listeners
        vnode, // new parent vnode
        options.children // new children
    )
},
// ...
}

```

`prepatch` 核心逻辑就是执行 `updateChildComponent` 方法，它的定义在 `src/core/instance/lifecycle.js` 中：

```

export function updateChildComponent (
  vm: Component,
  propsData: ?Object,
  listeners: ?Object,
  parentVnode: MountedComponentVNode,
  renderChildren: ?Array<VNode>
) {
  const hasChildren = !!((
    renderChildren ||
    vm.$options._renderChildren ||
    parentVnode.data.scopedSlots ||
    vm.$scopedSlots !== emptyObject
  ))

  // ...
  if (hasChildren) {
    vm.$slots = resolveSlots(renderChildren, parentVnode.context)
    vm.$forceUpdate()
  }
}

```

`updateChildComponent` 方法主要是去更新组件实例的一些属性，这里我们重点关注一下 `slot` 部分，由于 `<keep-alive>` 组件本质上支持了 `slot`，所以它执行 `prepatch` 的时候，需要对自己的 `children`，也就是这些 `slots` 做重新解析，并触发 `<keep-alive>` 组件实例 `$forceUpdate` 逻辑，也就是重新执行 `<keep-alive>` 的 `render` 方法，这个时候如果它包裹的第一个组件 `vnode` 命中缓存，则直接返回缓存中的 `vnode.componentInstance`，在我们的例子中就是缓存的 `A` 组件，接着又会执行 `patch` 过程，再次执行到 `createComponent` 方法，我们再回顾一下：

```

function createComponent (vnode, insertedVnodeQueue, parentElm, refElm) {
  let i = vnode.data
  if (isDef(i)) {
    const isReactivated = isDef(vnode.componentInstance) && i.keepAlive
    if (isDef(i = i.hook) && isDef(i = i.init)) {
      i(vnode, false /* hydrating */)
    }
    // after calling the init hook, if the vnode is a child component
    // it should've created a child instance and mounted it. the child
  }
}

```

```

    // component also has set the placeholder vnode's elm.
    // in that case we can just return the element and be done.
    if (isDef(vnode.componentInstance)) {
      initComponent(vnode, insertedVnodeQueue)
      insert(parentElm, vnode.elm, refElm)
      if (isTrue(isReactivated)) {
        reactivateComponent(vnode, insertedVnodeQueue, parentElm, refElm)
      }
      return true
    }
  }
}

```

这个时候 `isReactivated` 为 `true`, 并且在执行 `init` 钩子函数的时候不会再执行组件的 `mount` 过程了, 相关逻辑在 `src/core/vdom/create-component.js` 中:

```

const componentVNodeHooks = {
  init (vnode: VNodeWithData, hydrating: boolean): ?boolean {
    if (
      vnode.componentInstance &&
      !vnode.componentInstance._isDestroyed &&
      vnode.data.keepAlive
    ) {
      // kept-alive components, treat as a patch
      const mountedNode: any = vnode // work around flow
      componentVNodeHooks.prepatch(mountedNode, mountedNode)
    } else {
      const child = vnode.componentInstance = createComponentInstanceForVnode(
        vnode,
        activeInstance
      )
      child.$mount(hydrating ? vnode.elm : undefined, hydrating)
    }
  },
  // ...
}

```

这也就是被 `<keep-alive>` 包裹的组件在有缓存的时候就不会在执行组件的 `created`、`mounted` 等钩子函数的原因了。回到 `createComponent` 方法, 在 `isReactivated` 为 `true` 的情况下会执行 `reactivateComponent` 方法:

```

function reactivateComponent (vnode, insertedVnodeQueue, parentElm, refElm) {
  let i
  // hack for #4339: a reactivated component with inner transition
  // does not trigger because the inner node's created hooks are not called
  // again. It's not ideal to involve module-specific logic in here but
  // there doesn't seem to be a better way to do it.
  let innerNode = vnode
  while (innerNode.componentInstance) {

```

```

innerNode = innerNode.componentInstance._vnode
if (isDef(i = innerNode.data) && isDef(i = i.transition)) {
  for (i = 0; i < cbs.activate.length; ++i) {
    cbs.activate[i](emptyNode, innerNode)
  }
  insertedVnodeQueue.push(innerNode)
  break
}
// unlike a newly created component,
// a reactivated keep-alive component doesn't insert itself
insert(parentElm, vnode.elm, refElm)
}

```

前面部分的逻辑是解决对 `reactived` 组件 `transition` 动画不触发的问题，可以先不关注，最后通过执行 `insert(parentElm, vnode.elm, refElm)` 就把缓存的 DOM 对象直接插入到目标元素中，这样就完成了在数据更新的情况下的渲染过程。

生命周期

之前我们提到，组件一旦被 `<keep-alive>` 缓存，那么再次渲染的时候就不会执行 `created`、`mounted` 等钩子函数，但是我们很多业务场景都是希望在我们被缓存的组件再次被渲染的时候做一些事情，好在 Vue 提供了 `activated` 钩子函数，它的执行时机是 `<keep-alive>` 包裹的组件渲染的时候，接下来我们从源码角度来分析一下它的实现原理。

在渲染的最后一步，会执行 `invokeInsertHook(vnode, insertedVnodeQueue, isInitialPatch)` 函数执行 `vnode` 的 `insert` 钩子函数，它的定义在 `src/core/vdom/create-component.js` 中：

```

const componentVNodeHooks = {
  insert (vnode: MountedComponentVNode) {
    const { context, componentInstance } = vnode
    if (!componentInstance._isMounted) {
      componentInstance._isMounted = true
      callHook(componentInstance, 'mounted')
    }
    if (vnode.data.keepAlive) {
      if (context._isMounted) {
        // vue-router#1212
        // During updates, a kept-alive component's child components may
        // change, so directly walking the tree here may call activated hooks
        // on incorrect children. Instead we push them into a queue which will
        // be processed after the whole patch process ended.
        queueActivatedComponent(componentInstance)
      } else {
        activateChildComponent(componentInstance, true /* direct */)
      }
    }
  },
},

```

```
// ...
}
```

这里判断如果是被 `<keep-alive>` 包裹的组件已经 `mounted`，那么则执行 `queueActivatedComponent(componentInstance)`，否则执行 `activateChildComponent(componentInstance, true)`。我们先分析非 `mounted` 的情况，`activateChildComponent` 的定义在 `src/core/instance/lifecycle.js` 中：

```
export function activateChildComponent (vm: Component, direct?: boolean) {
  if (direct) {
    vm._directInactive = false
    if (isInInactiveTree(vm)) {
      return
    }
  } else if (vm._directInactive) {
    return
  }
  if (vm._inactive || vm._inactive === null) {
    vm._inactive = false
    for (let i = 0; i < vm.$children.length; i++) {
      activateChildComponent(vm.$children[i])
    }
    callHook(vm, 'activated')
  }
}
```

可以看到这里就是执行组件的 `acitvated` 钩子函数，并且递归去执行它的所有子组件的 `activated` 钩子函数。

那么再看 `queueActivatedComponent` 的逻辑，它定义在 `src/core/observer/scheduler.js` 中：

```
export function queueActivatedComponent (vm: Component) {
  vm._inactive = false
  activatedChildren.push(vm)
}
```

这个逻辑很简单，把当前 `vm` 实例添加到 `activatedChildren` 数组中，等所有的渲染完毕，在 `nextTick` 后会执行 `flushSchedulerQueue`，这个时候就会执行：

```
function flushSchedulerQueue () {
  // ...
  const activatedQueue = activatedChildren.slice()
  callActivatedHooks(activatedQueue)
  // ...
}

function callActivatedHooks (queue) {
  for (let i = 0; i < queue.length; i++) {
```

```

        queue[i]._inactive = true
        activateChildComponent(queue[i], true)  }
    }
}

```

也就是遍历所有的 `activatedChildren`，执行 `activateChildComponent` 方法，通过队列调的方式就是把整个 `activated` 时机延后了。

有 `activated` 钩子函数，也就有对应的 `deactivated` 钩子函数，它是发生在 `vnode` 的 `destory` 钩子函数，定义在 `src/core/vdom/create-component.js` 中：

```

const componentVNodeHooks = {
  destroy (vnode: MountedComponentVNode) {
    const { componentInstance } = vnode
    if (!componentInstance._isDestroyed) {
      if (!vnode.data.keepAlive) {
        componentInstance.$destroy()
      } else {
        deactivateChildComponent(componentInstance, true /* direct */)
      }
    }
  }
}

```

对于 `<keep-alive>` 包裹的组件而言，它会执行 `deactivateChildComponent(componentInstance, true)` 方法，定义在 `src/core/instance/lifecycle.js` 中：

```

export function deactivateChildComponent (vm: Component, direct?: boolean) {
  if (direct) {
    vm._directInactive = true
    if (isInInactiveTree(vm)) {
      return
    }
  }
  if (!vm._inactive) {
    vm._inactive = true
    for (let i = 0; i < vm.$children.length; i++) {
      deactivateChildComponent(vm.$children[i])
    }
    callHook(vm, 'deactivated')
  }
}

```

和 `activateChildComponent` 方法类似，就是执行组件的 `deactivated` 钩子函数，并且递归去执行它的所有子组件的 `deactivated` 钩子函数。

总结

那么至此，`<keep-alive>` 的实现原理就介绍完了，通过分析我们知道了 `<keep-alive>` 组件是一个抽象组件，它的实现通过自定义 `render` 函数并且利用了插槽，并且知道了 `<keep-alive>` 缓存 `vnode`，了解组件包裹的子元素——也就是插槽是如何做更新的。且在 `patch` 过程中对于已缓存的组件不会执行 `mounted`，所以不会有一般的组件的生命周期函数但是又提供了 `activated` 和 `deactivated` 钩子函数。另外我们还知道了 `<keep-alive>` 的 `props` 除了 `include` 和 `exclude` 还有文档中没有提到的 `max`，它能控制我们缓存的个数。

transition

在我们平时的前端项目开发中，经常会遇到如下需求，一个 DOM 节点的插入和删除或者是显示和隐藏，我们不想让它特别生硬，通常会考虑加一些过渡效果。

Vue.js 除了实现了强大的数据驱动，组件化的能力，也给我们提供了一整套过渡的解决方案。它内置了 `<transition>` 组件，我们可以利用它配合一些 CSS3 样式很方便地实现过渡动画，也可以利用它配合 JavaScript 的钩子函数实现过渡动画，在下列情形中，可以给任何元素和组件添加 `entering/leaving` 过渡：

- 条件渲染 (使用 `v-if`)
- 条件展示 (使用 `v-show`)
- 动态组件
- 组件根节点

那么举一个最简单的实例，如下：

```
let vm = new Vue({
  el: '#app',
  template: '<div id="demo">' +
    '<button v-on:click="show = !show">' +
    'Toggle' +
    '</button>' +
    '<transition :appear="true" name="fade">' +
    '<p v-if="show">hello</p>' +
    '</transition>' +
    '</div>',
  data() {
    return {
      show: true
    }
  }
})
```

```
.fade-enter-active, .fade-leave-active {
  transition: opacity .5s;
}
.fade-enter, .fade-leave-to {
  opacity: 0;
}
```

当我们点击按钮切换显示状态的时候，被 `<transition>` 包裹的内容会有过渡动画。那么接下来我们从源码的角度来分析它的实现原理。

内置组件

`<transition>` 组件和 `<keep-alive>` 组件一样，都是 Vue 的内置组件，而 `<transition>` 的定义在 `src/platforms/web/runtime/component/transition.js` 中，之所以在这里定义，是因为 `<transition>` 组件是 web 平台独有的，先来看一下它的实现：

```

export default {
  name: 'transition',
  props: transitionProps,
  abstract: true,

  render (h: Function) {
    let children: any = this.$slots.default
    if (!children) {
      return
    }

    // filter out text nodes (possible whitespaces)
    children = children.filter((c: VNode) => c.tag || isAsyncPlaceholder(c))
    /* istanbul ignore if */
    if (!children.length) {
      return
    }

    // warn multiple elements
    if (process.env.NODE_ENV !== 'production' && children.length > 1) {
      warn(
        '<transition> can only be used on a single element. Use ' +
        '<transition-group> for lists.',
        this.$parent
      )
    }

    const mode: string = this.mode

    // warn invalid mode
    if (process.env.NODE_ENV !== 'production' &&
      mode && mode !== 'in-out' && mode !== 'out-in'
    ) {
      warn(
        'invalid <transition> mode: ' + mode,
        this.$parent
      )
    }

    const rawChild: VNode = children[0]

    // if this is a component root node and the component's
    // parent container node also has transition, skip.
    if (hasParentTransition(this.$vnode)) {
      return rawChild
    }
  }
}

```

```

// apply transition data to child
// use getRealChild() to ignore abstract components e.g. keep-alive
const child: ?VNode = getRealChild(rawChild)
/* istanbul ignore if */
if (!child) {
  return rawChild
}

if (this._leaving) {
  return placeholder(h, rawChild)
}

// ensure a key that is unique to the vnode type and to this transition
// component instance. This key will be used to remove pending leaving nodes
// during entering.
const id: string = `__transition-${this._uid}-`
child.key = child.key == null
  ? child.isComment
    ? id + 'comment'
    : id + child.tag
  : isPrimitive(child.key)
  ? (String(child.key).indexOf(id) === 0 ? child.key : id + child.key)
  : child.key

const data: Object = (child.data || (child.data = {})).transition = extractTransitionData(this)
const oldRawChild: VNode = this._vnode
const oldChild: VNode = getRealChild(oldRawChild)

// mark v-show
// so that the transition module can hand over the control to the directive
if (child.data.directives && child.data.directives.some(d => d.name === 'show'))
) {
  child.data.show = true
}

if (
  oldChild &&
  oldChild.data &&
  !isSameChild(child, oldChild) &&
  !isAsyncPlaceholder(oldChild) &&
  // #6687 component root is a comment node
  !(oldChild.componentInstance && oldChild.componentInstance._vnode.isComment)
) {
  // replace old child transition data with fresh one
  // important for dynamic transitions!
  const oldData: Object = oldChild.data.transition = extend({}, data)
  // handle transition mode
  if (mode === 'out-in') {
    // return placeholder node and queue update when leave finishes
}

```

```

        this._leaving = true
        mergeVNodeHook(oldData, 'afterLeave', () => {
            this._leaving = false
            this.$forceUpdate()
        })
        return placeholder(h, rawChild)
    } else if (mode === 'in-out') {
        if (isAsyncPlaceholder(child)) {
            return oldRawChild
        }
        let delayedLeave
        const performLeave = () => { delayedLeave() }
        mergeVNodeHook(data, 'afterEnter', performLeave)
        mergeVNodeHook(data, 'enterCancelled', performLeave)
        mergeVNodeHook(oldData, 'delayLeave', leave => { delayedLeave = leave })
    }
}

return rawChild
}
}

```

`<transition>` 组件和 `<keep-alive>` 组件有几点实现类似，同样是抽象组件，同样直接实现 `render` 函数，同样利用了默认插槽。`<transition>` 组件非常灵活，支持的 `props` 非常多：

```

export const transitionProps = {
    name: String,
    appear: Boolean,
    css: Boolean,
    mode: String,
    type: String,
    enterClass: String,
    leaveClass: String,
    enterToClass: String,
    leaveToClass: String,
    enterActiveClass: String,
    leaveActiveClass: String,
    appearClass: String,
    appearActiveClass: String,
    appearToClass: String,
    duration: [Number, String, Object]
}

```

这些配置我们稍后会分析它们的作用，`<transition>` 组件另一个重要的就是 `render` 函数的实现，`render` 函数主要作用就是渲染生成 `vnode`，下面来看一下这部分的逻辑。

- 处理 `children`

```

let children: any = this.$slots.default

```

```

if (!children) {
  return
}

// filter out text nodes (possible whitespaces)
children = children.filter((c: VNode) => c.tag || isAsyncPlaceholder(c))
/* istanbul ignore if */
if (!children.length) {
  return
}

// warn multiple elements
if (process.env.NODE_ENV !== 'production' && children.length > 1) {
  warn(
    '<transition> can only be used on a single element. Use ' +
    '<transition-group> for lists.',
    this.$parent
  )
}

```

先从默认插槽中获取 `<transition>` 包裹的子节点，并且判断了子节点的长度，如果长度为 0，则直接返回，否则判断长度如果大于 1，也会在开发环境报警告，因为 `<transition>` 组件是只能包裹一个子节点的。

- 处理 `model`

```

const mode: string = this.mode

// warn invalid mode
if (process.env.NODE_ENV !== 'production' &&
  mode && mode !== 'in-out' && mode !== 'out-in'
) {
  warn(
    'invalid <transition> mode: ' + mode,
    this.$parent
  )
}

```

过渡组件对 `mode` 的支持只有 2 种，`in-out` 或者是 `out-in`。

- 获取 `rawChild` & `child`

```

const rawChild: VNode = children[0]

// if this is a component root node and the component's
// parent container node also has transition, skip.
if (hasParentTransition(this.$vnode)) {
  return rawChild
}

```

```
// apply transition data to child
// use getRealChild() to ignore abstract components e.g. keep-alive
const child: ?VNode = getRealChild(rawChild)
/* istanbul ignore if */
if (!child) {
  return rawChild
}
```

`rawChild` 就是第一个子节点 `vnode`，接着判断当前 `<transition>` 如果是组件根节点并且外面包裹该组件的容器也是 `<transition>` 的时候要跳过。来看一下 `hasParentTransition` 的实现：

```
function hasParentTransition (vnode: VNode): ?boolean {
  while ((vnode = vnode.parent)) {
    if (vnode.data.transition) {
      return true
    }
  }
}
```

因为传入的是 `this.$vnode`，也就是 `<transition>` 组件的占位 `vnode`，只有当它同时作为根 `vnode`，也就是 `vm._vnode` 的时候，它的 `parent` 才不会为空，并且判断 `parent` 也是 `<transition>` 组件，才返回 `true`，`vnode.data.transition` 我们稍后会介绍。

`getRealChild` 的目的是获取组件的非抽象子节点，因为 `<transition>` 很可能会包裹一个 `keep-alive`，它的实现如下：

```
// in case the child is also an abstract component, e.g. <keep-alive>
// we want to recursively retrieve the real component to be rendered
function getRealChild (vnode: ?VNode): ?VNode {
  const compOptions: ?VNodeComponentOptions = vnode && vnode.componentOptions
  if (compOptions && compOptions.Ctor.options.abstract) {
    return getRealChild(getFirstComponentChild(compOptions.children))
  } else {
    return vnode
  }
}
```

会递归找到第一个非抽象组件的 `vnode` 并返回，在我们这个 case 下，`rawChild === child`。

- 处理 `id` & `data`

```
// ensure a key that is unique to the vnode type and to this transition
// component instance. This key will be used to remove pending leaving nodes
// during entering.
const id: string = `__transition-${this._uid}-`
child.key = child.key == null
  ? child.isComment
```

```

    ? id + 'comment'
    : id + child.tag
: isPrimitive(child.key)
? (String(child.key).indexOf(id) === 0 ? child.key : id + child.key)
: child.key

const data: Object = (child.data || (child.data = {})).transition = extractTransitionData(this)
const oldRawChild: VNode = this._vnode
const oldChild: VNode = getRealChild(oldRawChild)

// mark v-show
// so that the transition module can hand over the control to the directive
if (child.data.directives && child.data.directives.some(d => d.name === 'show')) {
  child.data.show = true
}

```

先根据 `key` 等一系列条件获取 `id`，接着从当前通过 `extractTransitionData` 组件实例上提取出过渡所需要的数据：

```

export function extractTransitionData (comp: Component): Object {
  const data = {}
  const options: ComponentOptions = comp.$options
  // props
  for (const key in options.propsData) {
    data[key] = comp[key]
  }
  // events.
  // extract listeners and pass them directly to the transition methods
  const listeners: ?Object = options._parentListeners
  for (const key in listeners) {
    data[camelize(key)] = listeners[key]
  }
  return data
}

```

首先是遍历 `props` 赋值到 `data` 中，接着是遍历所有父组件的事件也把事件回调赋值到 `data` 中。

这样 `child.data.transition` 中就包含了过渡所需的一些数据，这些稍后都会用到，对于 `child` 如果使用了 `v-show` 指令，也会把 `child.data.show` 设置为 `true`，在我们的例子中，得到的 `child.data` 如下：

```
{
  transition: {
    appear: true,
    name: 'fade'
  }
}
```

至于 `oldRawChild` 和 `oldChild` 是与后面的判断逻辑相关，这些我们这里先不介绍。

transition module

刚刚我们介绍完 `<transition>` 组件的实现，它的 `render` 阶段只获取了一些数据，并且返回了渲染的 `vnode`，并没有任何和动画相关，而动画相关的逻辑全部在

`src/platforms/web/modules/transition.js` 中：

```
function _enter (_: any, vnode: VNodeWithData) {
  if (vnode.data.show !== true) {
    enter(vnode)
  }
}

export default inBrowser ? {
  create: _enter,
  activate: _enter,
  remove (vnode: VNode, rm: Function) {
    /* istanbul ignore else */
    if (vnode.data.show !== true) {
      leave(vnode, rm)
    } else {
      rm()
    }
  }
} : {}
```

在之前介绍事件实现的章节中我们提到过在 `vnode patch` 的过程中，会执行很多钩子函数，那么对于过渡的实现，它只接收了 `create` 和 `activate` 2个钩子函数，我们知道 `create` 钩子函数只有当节点的创建过程才会执行，而 `remove` 会在节点销毁的时候执行，这也就印证了 `<transition>` 必须要满足 `v-if`、动态组件、组件根节点条件之一了，对于 `v-show` 在它的指令的钩子函数中也会执行相关逻辑，这块儿先不介绍。

过渡动画提供了2个时机，一个是 `create` 和 `activate` 的时候提供了 `entering` 进入动画，一个是 `remove` 的时候提供了 `leaving` 离开动画，那么接下来我们就来分别去分析这两个过程。

entering

整个 `entering` 过程的实现是 `enter` 函数：

```
export function enter (vnode: VNodeWithData, toggleDisplay: ?() => void) {
  const el: any = vnode.elm

  // call leave callback now
  if (isDef(el._leaveCb)) {
    el._leaveCb.cancelled = true
```

```
    el._leaveCb()
}

const data = resolveTransition(vnode.data.transition)
if (isUndef(data)) {
  return
}

/* istanbul ignore if */
if (isDef(el._enterCb) || el.nodeType !== 1) {
  return
}

const {
  css,
  type,
  enterClass,
  enterToClass,
  enterActiveClass,
  appearClass,
  appearToClass,
  appearActiveClass,
  beforeEnter,
  enter,
  afterEnter,
  enterCancelled,
  beforeAppear,
  appear,
  afterAppear,
  appearCancelled,
  duration
} = data

// activeInstance will always be the <transition> component managing this
// transition. One edge case to check is when the <transition> is placed
// as the root node of a child component. In that case we need to check
// <transition>'s parent for appear check.
let context = activeInstance
let transitionNode = activeInstance.$vnode
while (transitionNode && transitionNode.parent) {
  transitionNode = transitionNode.parent
  context = transitionNode.context
}

const isAppear = !context._isMounted || !vnode.isRootInsert

if (isAppear && !appear && appear !== '') {
  return
}

const startClass = isAppear && appearClass
```

```

    ? appearClass
    : enterClass
  const activeClass = isAppear && appearActiveClass
    ? appearActiveClass
    : enterActiveClass
  const toClass = isAppear && appearToClass
    ? appearToClass
    : enterToClass

  const beforeEnterHook = isAppear
    ? (beforeAppear || beforeEnter)
    : beforeEnter
  const enterHook = isAppear
    ? (typeof appear === 'function' ? appear : enter)
    : enter
  const afterEnterHook = isAppear
    ? (afterAppear || afterEnter)
    : afterEnter
  const enterCancelledHook = isAppear
    ? (appearCancelled || enterCancelled)
    : enterCancelled

  const explicitEnterDuration: any = toNumber(
    isObject(duration)
      ? duration.enter
      : duration
  )

  if (process.env.NODE_ENV !== 'production' && explicitEnterDuration != null) {
    checkDuration(explicitEnterDuration, 'enter', vnode)
  }

  const expectsCSS = css !== false && !isIE9
  const userWantsControl = getHookArgumentsLength(enterHook)

  const cb = el._enterCb = once(() => {
    if (expectsCSS) {
      removeTransitionClass(el, toClass)
      removeTransitionClass(el, activeClass)
    }
    if (cb.cancelled) {
      if (expectsCSS) {
        removeTransitionClass(el, startClass)
      }
      enterCancelledHook && enterCancelledHook(el)
    } else {
      afterEnterHook && afterEnterHook(el)
    }
    el._enterCb = null
  })

```

```

if (!vnode.data.show) {
  // remove pending leave element on enter by injecting an insert hook
  mergeVNodeHook(vnode, 'insert', () => {
    const parent = el.parentNode
    const pendingNode = parent && parent._pending && parent._pending[vnode.key]
    if (pendingNode &&
        pendingNode.tag === vnode.tag &&
        pendingNode.elm._leaveCb)
      pendingNode.elm._leaveCb()
    }
    enterHook && enterHook(el, cb)
  })
}

// start enter transition
beforeEnterHook && beforeEnterHook(el)
if (expectsCSS) {
  addTransitionClass(el, startClass)
  addTransitionClass(el, activeClass)
  nextFrame(() => {
    removeTransitionClass(el, startClass)
    if (!cb.cancelled) {
      addTransitionClass(el, toClass)
      if (!userWantsControl) {
        if (isValidDuration(explicitEnterDuration)) {
          setTimeout(cb, explicitEnterDuration)
        } else {
          whenTransitionEnds(el, type, cb)
        }
      }
    }
  })
}

if (vnode.data.show) {
  toggleDisplay && toggleDisplay()
  enterHook && enterHook(el, cb)
}

if (!expectsCSS && !userWantsControl) {
  cb()
}
}

```

`enter` 的代码很长，我们先分析其中的核心逻辑。

- 解析过渡数据

```
const data = resolveTransition(vnode.data.transition)
```

```

if (isUndef(data)) {
  return
}

const {
  css,
  type,
  enterClass,
  enterToClass,
  enterActiveClass,
  appearClass,
  appearToClass,
  appearActiveClass,
  beforeEnter,
  enter,
  afterEnter,
  enterCancelled,
  beforeAppear,
  appear,
  afterAppear,
  appearCancelled,
  duration
} = data

```

从 `vnode.data.transition` 中解析出过渡相关的一些数据，`resolveTransition` 的定义在 `src/platforms/web/transition-util.js` 中：

```

export function resolveTransition (def?: string | Object): ?Object {
  if (!def) {
    return
  }
  /* istanbul ignore else */
  if (typeof def === 'object') {
    const res = {}
    if (def.css !== false) {
      extend(res, autoCssTransition(def.name || 'v'))
    }
    extend(res, def)
    return res
  } else if (typeof def === 'string') {
    return autoCssTransition(def)
  }
}

const autoCssTransition: (name: string) => Object = cached(name => {
  return {
    enterClass: `${name}-enter`,
    enterToClass: `${name}-enter-to`,
    enterActiveClass: `${name}-enter-active`,
    leaveClass: `${name}-leave`,
  }
})

```

```

        leaveToClass: `${name}-leave-to`,
        leaveActiveClass: `${name}-leave-active`
    }
})

```

`resolveTransition` 会通过 `autoCssTransition` 处理 `name` 属性，生成一个用来描述各个阶段的 `class` 名称的对象，扩展到 `def` 中并返回给 `data`，这样我们就可以从 `data` 中获取到过渡相关的所有数据。

- 处理边界情况

```

// activeInstance will always be the <transition> component managing this
// transition. One edge case to check is when the <transition> is placed
// as the root node of a child component. In that case we need to check
// <transition>'s parent for appear check.
let context = activeInstance
let transitionNode = activeInstance.$vnode
while (transitionNode && transitionNode.parent) {
    transitionNode = transitionNode.parent
    context = transitionNode.context
}

const isAppear = !context._isMounted || !vnode.isRootInsert

if (isAppear && !appear && appear !== '!') {
    return
}

```

这是为了处理当 `<transition>` 作为子组件的根节点，那么我们需要检查它的父组件作为 `appear` 的检查。`isAppear` 表示当前上下文实例还没有 `mounted`，第一次出现的时机。如果是第一次并且 `<transition>` 组件没有配置 `appear` 的话，直接返回。

- 定义过渡类名、钩子函数和其它配置

```

const startClass = isAppear && appearClass
    ? appearClass
    : enterClass
const activeClass = isAppear && appearActiveClass
    ? appearActiveClass
    : enterActiveClass
const toClass = isAppear && appearToClass
    ? appearToClass
    : enterToClass

const beforeEnterHook = isAppear
    ? (beforeAppear || beforeEnter)
    : beforeEnter
const enterHook = isAppear
    ? (typeof appear === 'function' ? appear : enter)

```

```

    : enter
  const afterEnterHook = isAppear
    ? (afterAppear || afterEnter)
    : afterEnter
  const enterCancelledHook = isAppear
    ? (appearCancelled || enterCancelled)
    : enterCancelled

  const explicitEnterDuration: any = toNumber(
    isObject(duration)
      ? duration.enter
      : duration
  )

  if (process.env.NODE_ENV !== 'production' && explicitEnterDuration != null) {
    checkDuration(explicitEnterDuration, 'enter', vnode)
  }

  const expectsCSS = css !== false && !isIE9
  const userWantsControl = getHookArgumentsLength(enterHook)

  const cb = el._enterCb = once(() => {
    if (expectsCSS) {
      removeTransitionClass(el, toClass)
      removeTransitionClass(el, activeClass)
    }
    if (cb.cancelled) {
      if (expectsCSS) {
        removeTransitionClass(el, startClass)
      }
      enterCancelledHook && enterCancelledHook(el)
    } else {
      afterEnterHook && afterEnterHook(el)
    }
    el._enterCb = null
  })
}

```

对于过渡类名方面，`startClass` 定义进入过渡的开始状态，在元素被插入时生效，在下一个帧移除；`activeClass` 定义过渡的状态，在元素整个过渡过程中作用，在元素被插入时生效，在`transition/animation` 完成之后移除；`toClass` 定义进入过渡的结束状态，在元素被插入一帧后生效（与此同时 `startClass` 被删除），在 `<transition>/animation` 完成之后移除。

对于过渡钩子函数方面，`beforeEnterHook` 是过渡开始前执行的钩子函数，`enterHook` 是在元素插入后或者是 `v-show` 显示切换后执行的钩子函数。`afterEnterHook` 是在过渡动画执行完后的钩子函数。

`explicitEnterDuration` 表示 enter 动画执行的时间。

`expectsCSS` 表示过渡动画是受 CSS 的影响。

`cb` 定义的是过渡完成执行的回调函数。

- 合并 `insert` 钩子函数

```
if (!vnode.data.show) {
  // remove pending leave element on enter by injecting an insert hook
  mergeVNodeHook(vnode, 'insert', () => {
    const parent = el.parentNode
    const pendingNode = parent && parent._pending && parent._pending[vnode.key]
    if (pendingNode &&
        pendingNode.tag === vnode.tag &&
        pendingNode.elm._leaveCb)
      pendingNode.elm._leaveCb()
    enterHook && enterHook(el, cb)
  })
}
```

`mergeVNodeHook` 的定义在 `src/core/vdom/helpers/merge-hook.js` 中：

```
export function mergeVNodeHook (def: Object, hookKey: string, hook: Function) {
  if (def instanceof VNode) {
    def = def.data.hook || (def.data.hook = {})
  }
  let invoker
  const oldHook = def[hookKey]

  function wrappedHook () {
    hook.apply(this, arguments)
    // important: remove merged hook to ensure it's called only once
    // and prevent memory leak
    remove(invoker.fns, wrappedHook)
  }

  if (isUndef(oldHook)) {
    // no existing hook
    invoker = createFnInvoker([wrappedHook])
  } else {
    /* istanbul ignore if */
    if (isDef(oldHook.fns) && isTrue(oldHook.merged)) {
      // already a merged invoker
      invoker = oldHook
      invoker.fns.push(wrappedHook)
    } else {
      // existing plain hook
      invoker = createFnInvoker([oldHook, wrappedHook])
    }
  }
}
```

```

    invoker.merged = true
    def[hookKey] = invoker
}

```

`mergeVNodeHook` 的逻辑很简单，就是把 `hook` 函数合并到 `def.data.hook[hookKey]` 中，生成新的 `invoker`，`createFnInvoker` 方法我们在分析事件章节的时候已经介绍过了。

我们之前知道组件的 `vnode` 原本定义了 `init`、`prepatch`、`insert`、`destroy` 四个钩子函数，而 `mergeVNodeHook` 函数就是把一些新的钩子函数合并进来，例如在 `<transition>` 过程中合并的 `insert` 钩子函数，就会合并到组件 `vnode` 的 `insert` 钩子函数中，这样当组件插入后，就会执行我们定义的 `enterHook` 了。

- 开始执行过渡动画

```

// start enter transition
beforeEnterHook && beforeEnterHook(el)
if (expectsCSS) {
  addTransitionClass(el, startClass)
  addTransitionClass(el, activeClass)
  nextFrame(() => {
    removeTransitionClass(el, startClass)
    if (!cb.cancelled) {
      addTransitionClass(el, toClass)
      if (!userWantsControl) {
        if (isValidDuration(explicitEnterDuration)) {
          setTimeout(cb, explicitEnterDuration)
        } else {
          whenTransitionEnds(el, type, cb)
        }
      }
    }
  })
}

```

首先执行 `beforeEnterHook` 钩子函数，把当前元素的 DOM 节点 `el` 传入，然后判断 `expectsCSS`，如果为 `true` 则表明希望用 CSS 来控制动画，那么会执行 `addTransitionClass(el, startClass)` 和 `addTransitionClass(el, activeClass)`，它的定义在 `src/platforms/runtime/transition-util.js` 中：

```

export function addTransitionClass (el: any, cls: string) {
  const transitionClasses = el._transitionClasses || (el._transitionClasses = [])
  if (transitionClasses.indexOf(cls) < 0) {
    transitionClasses.push(cls)
    addClass(el, cls)
  }
}

```

其实非常简单，就是给当前 DOM 元素 `el` 添加样式 `cls`，所以这里添加了 `startClass` 和 `activeClass`，在我们的例子中就是给 `p` 标签添加了 `fade-enter` 和 `fade-enter-active` 2 个样式。

接下来执行了 `nextFrame`：

```
const raf = inBrowser
? window.requestAnimationFrame
? window.requestAnimationFrame.bind(window)
: setTimeout
: fn => fn()

export function nextFrame (fn: Function) {
  raf(() => {
    raf(fn)
  })
}
```

它就是一个简单的 `requestAnimationFrame` 的实现，它的参数 `fn` 会在下一帧执行，因此下一帧执行了 `removeTransitionClass(el, startClass)`：

```
export function removeTransitionClass (el: any, cls: string) {
  if (el._transitionClasses) {
    remove(el._transitionClasses, cls)
  }
  removeClass(el, cls)
}
```

把 `startClass` 移除，在我们的等例子中就是移除 `fade-enter` 样式。然后判断此时过渡没有被取消，则执行 `addTransitionClass(el, toClass)` 添加 `toClass`，在我们的例子中就是添加了 `fade-enter-to`。然后判断 `!userWantsControl`，也就是用户不通过 `enterHook` 钩子函数控制动画，这时候如果用户指定了 `explicitEnterDuration`，则延时这个时间执行 `cb`，否则通过 `whenTransitionEnds(el, type, cb)` 决定执行 `cb` 的时机：

```
export function whenTransitionEnds (
  el: Element,
  expectedType: ?string,
  cb: Function
) {
  const { type, timeout, propCount } = getTransitionInfo(el, expectedType)
  if (!type) return cb()
  const event: string = type === <transition> ? transitionEndEvent : animationEndEvent
  let ended = 0
  const end = () => {
    el.removeEventListener(event, onEnd)
    cb()
  }
}
```

```

const onEnd = e => {
  if (e.target === el) {
    if (++ended >= propCount) {
      end()
    }
  }
}
setTimeout(() => {
  if (ended < propCount) {
    end()
  }
}, timeout + 1)
el.addEventListener(event, onEnd)
}

```

`whenTransitionEnds` 的逻辑具体不深讲了，本质上就利用了过渡动画的结束事件来决定 `cb` 函数的执行。

最后再回到 `cb` 函数：

```

const cb = el._enterCb = once(() => {
  if (expectsCSS) {
    removeTransitionClass(el, toClass)
    removeTransitionClass(el, activeClass)
  }
  if (cb.cancelled) {
    if (expectsCSS) {
      removeTransitionClass(el, startClass)
    }
    enterCancelledHook && enterCancelledHook(el)
  } else {
    afterEnterHook && afterEnterHook(el)
  }
  el._enterCb = null
})

```

其实很简单，执行了 `removeTransitionClass(el, toClass)` 和 `removeTransitionClass(el, activeClass)` 把 `toClass` 和 `activeClass` 移除，然后判断如果有取消，如果取消则移除 `startClass` 并执行 `enterCancelledHook`，否则执行 `afterEnterHook(el)`。

那么到这里，`entering` 的过程就介绍完了。

leaving

与 `entering` 相对的就是 `leaving` 阶段了，`entering` 主要发生在组件插入后，而 `leaving` 主要发生在组件销毁前。

```

export function leave (vnode: VNodeWithData, rm: Function) {

```

```

const el: any = vnode.elm

// call enter callback now
if (isDef(el._enterCb)) {
  el._enterCb.cancelled = true
  el._enterCb()
}

const data = resolveTransition(vnode.data.transition)
if (isUndef(data) || el.nodeType !== 1) {
  return rm()
}

/* istanbul ignore if */
if (isDef(el._leaveCb)) {
  return
}

const {
  css,
  type,
  leaveClass,
  leaveToClass,
  leaveActiveClass,
  beforeLeave,
  leave,
  afterLeave,
  leaveCancelled,
  delayLeave,
  duration
} = data

const expectsCSS = css !== false && !isIE9
const userWantsControl = getHookArgumentsLength(leave)

const explicitLeaveDuration: any = toNumber(
  isObject(duration)
    ? duration.leave
    : duration
)

if (process.env.NODE_ENV !== 'production' && isDef(explicitLeaveDuration)) {
  checkDuration(explicitLeaveDuration, 'leave', vnode)
}

const cb = el._leaveCb = once(() => {
  if (el.parentNode && el.parentNode._pending) {
    el.parentNode._pending[vnode.key] = null
  }
  if (expectsCSS) {
    removeTransitionClass(el, leaveToClass)
  }
})

```

```

        removeTransitionClass(el, leaveActiveClass)
    }
    if (cb.cancelled) {
        if (expectsCSS) {
            removeTransitionClass(el, leaveClass)
        }
        leaveCancelled && leaveCancelled(el)
    } else {
        rm()
        afterLeave && afterLeave(el)
    }
    el._leaveCb = null
})

if (delayLeave) {
    delayLeave(performLeave)
} else {
    performLeave()
}

function performLeave () {
    // the delayed leave may have already been cancelled
    if (cb.cancelled) {
        return
    }
    // record leaving element
    if (!vnode.data.show) {
        (el.parentNode._pending || (el.parentNode._pending = {}))[(vnode.key: any)] =
vnode
    }
    beforeLeave && beforeLeave(el)
    if (expectsCSS) {
        addTransitionClass(el, leaveClass)
        addTransitionClass(el, leaveActiveClass)
        nextFrame(() => {
            removeTransitionClass(el, leaveClass)
            if (!cb.cancelled) {
                addTransitionClass(el, leaveToClass)
                if (!userWantsControl) {
                    if (isValidDuration(explicitLeaveDuration)) {
                        setTimeout(cb, explicitLeaveDuration)
                    } else {
                        whenTransitionEnds(el, type, cb)
                    }
                }
            }
        })
    }
    leave && leave(el, cb)
    if (!expectsCSS && !userWantsControl) {
        cb()
    }
}

```

```
        }
    }
}
```

纵观 `leave` 的实现，和 `enter` 的实现几乎是一个镜像过程，不同的是从 `data` 中解析出来的是 `leave` 相关的样式类名和钩子函数。还有一点不同是可以配置 `delayLeave`，它是一个函数，可以延时执行 `leave` 的相关过渡动画，在 `leave` 动画执行完后，它会执行 `rm` 函数把节点从 DOM 中真正做移除。

总结

那么到此为止基本的 `<transition>` 过渡的实现分析完毕了，总结起来，Vue 的过渡实现分为以下几个步骤：

1. 自动嗅探目标元素是否应用了 CSS 过渡或动画，如果是，在恰当的时机添加/删除 CSS 类名。
2. 如果过渡组件提供了 JavaScript 钩子函数，这些钩子函数将在恰当的时机被调用。
3. 如果没有找到 JavaScript 钩子并且也没有检测到 CSS 过渡/动画，DOM 操作(插入/删除)在下一帧中立即执行。

所以真正执行动画的是我们写的 CSS 或者是 JavaScript 钩子函数，而 Vue 的 `<transition>` 只是帮我们很好地管理了这些 CSS 的添加/删除，以及钩子函数的执行时机。

transition-group

前一节我们介绍了 `<transiiton>` 组件的实现原理，它只能针对单一元素实现过渡效果。我们做前端开发经常会遇到列表的需求，我们对列表元素进行添加和删除，有时候也希望有过渡效果，Vue.js 提供了 `<transition-group>` 组件，很好地帮助我们实现了列表的过渡效果。那么接下来我们就来分析一下它的实现原理。

为了更直观，我们也是通过一个示例来说明：

```
let vm = new Vue({
  el: '#app',
  template: '<div id="list-complete-demo" class="demo">' +
    '<button v-on:click="add">Add</button>' +
    '<button v-on:click="remove">Remove</button>' +
    '<transition-group name="list-complete" tag="p">' +
    '<span v-for="item in items" v-bind:key="item" class="list-complete-item">' +
    '{{ item }}' +
    '</span>' +
    '</transition-group>' +
    '</div>',
  data: {
    items: [1, 2, 3, 4, 5, 6, 7, 8, 9],
    nextNum: 10
  },
  methods: {
    randomIndex: function () {
      return Math.floor(Math.random() * this.items.length)
    },
    add: function () {
      this.items.splice(this.randomIndex(), 0, this.nextNum++)
    },
    remove: function () {
      this.items.splice(this.randomIndex(), 1)
    }
  }
})
```

```
.list-complete-item {
  display: inline-block;
  margin-right: 10px;
}
.list-complete-move {
  transition: all 1s;
}
.list-complete-enter, .list-complete-leave-to {
  opacity: 0;
  transform: translateY(30px);
```

```

    }
    .list-complete-enter-active {
      transition: all 1s;
    }
    .list-complete-leave-active {
      transition: all 1s;
      position: absolute;
    }
  
```

这个示例初始会展现 1-9 十个数字，当我们点击 `Add` 按钮时，会生成 `nextNum` 并随机在当前数列表中插入；当我们点击 `Remove` 按钮时，会随机删除掉一个数。我们会发现在数添加删除的过程中在列表中会有过渡动画，这就是 `<transition-group>` 组件配合我们定义的 CSS 产生的效果。

我们首先还是来分析 `<transition-group>` 组件的实现，它的定义在 `src/platforms/web/runtime/components/transitions.js` 中：

```

const props = extend({
  tag: String,
  moveClass: String
}, transitionProps)

delete props.mode

export default {
  props,

  beforeMount () {
    const update = this._update
    this._update = (vnode, hydrating) => {
      // force removing pass
      this.__patch__(
        this._vnode,
        this.kept,
        false, // hydrating
        true // removeOnly (!important, avoids unnecessary moves)
      )
      this._vnode = this.kept
      update.call(this, vnode, hydrating)
    }
  },
  render (h: Function) {
    const tag: string = this.tag || this.$vnode.data.tag || 'span'
    const map: Object = Object.create(null)
    const prevChildren: Array<VNode> = this.prevChildren = this.children
    const rawChildren: Array<VNode> = this.$slots.default || []
    const children: Array<VNode> = this.children = []
    const transitionData: Object = extractTransitionData(this)

    for (let i = 0; i < rawChildren.length; i++) {
  
```

```

    const c: VNode = rawChildren[i]
    if (c.tag) {
      if (c.key != null && String(c.key).indexOf('__vlist') !== 0) {
        children.push(c)
        map[c.key] = c
        ;(c.data || (c.data = {})).transition = transitionData
      } else if (process.env.NODE_ENV !== 'production') {
        const opts: ?VNodeComponentOptions = c.componentOptions
        const name: string = opts ? (opts.Ctor.options.name || opts.tag || '') :
          c.tag
        warn(`<transition-group> children must be keyed: <${name}>`)
      }
    }
  }

  if (prevChildren) {
    const kept: Array<VNode> = []
    const removed: Array<VNode> = []
    for (let i = 0; i < prevChildren.length; i++) {
      const c: VNode = prevChildren[i]
      c.data.transition = transitionData
      c.data.pos = c.elm.getBoundingClientRect()
      if (map[c.key]) {
        kept.push(c)
      } else {
        removed.push(c)
      }
    }
    this.kept = h(tag, null, kept)
    this.removed = removed
  }

  return h(tag, null, children)
},
updated () {
  const children: Array<VNode> = this.prevChildren
  const moveClass: string = this.moveClass || ((this.name || 'v') + '-move')
  if (!children.length || !this.hasMove(children[0].elm, moveClass)) {
    return
  }

  // we divide the work into three loops to avoid mixing DOM reads and writes
  // in each iteration - which helps prevent layout thrashing.
  children.forEach(callPendingCbs)
  children.forEach(recordPosition)
  children.forEach(applyTranslation)

  // force reflow to put everything in position
  // assign to this to avoid being removed in tree-shaking
  // $flow-disable-line
}

```

```

this._reflow = document.body.offsetHeight

children.forEach((c: VNode) => {
  if (c.data.moved) {
    var el: any = c.elm
    var s: any = el.style
    addTransitionClass(el, moveClass)
    s.transform = s.WebkitTransform = s.transitionDuration = ''
    el.addEventListener(transitionEndEvent, el._moveCb = function cb (e) {
      if (!e || /transform$/.test(e.propertyName)) {
        el.removeEventListener(transitionEndEvent, cb)
        el._moveCb = null
        removeTransitionClass(el, moveClass)
      }
    })
  }
},
methods: {
  hasMove (el: any, moveClass: string): boolean {
    /* istanbul ignore if */
    if (!hasTransition) {
      return false
    }
    /* istanbul ignore if */
    if (this._hasMove) {
      return this._hasMove
    }
    // Detect whether an element with the move class applied has
    // CSS transitions. Since the element may be inside an entering
    // transition at this very moment, we make a clone of it and remove
    // all other transition classes applied to ensure only the move class
    // is applied.
    const clone: HTMLElement = el.cloneNode()
    if (el._transitionClasses) {
      el._transitionClasses.forEach((cls: string) => { removeClass(clone, cls) })
    }
    addClass(clone, moveClass)
    clone.style.display = 'none'
    this.$el.appendChild(clone)
    const info: Object = getTransitionInfo(clone)
    this.$el.removeChild(clone)
    return (this._hasMove = info.hasTransform)
  }
}
}

```

render 函数

`<transition-group>` 组件也是由 `render` 函数渲染生成 `vnode`，接下来我们先分析 `render` 的实现。

- 定义一些变量

```
const tag: string = this.tag || this.$vnode.data.tag || 'span'
const map: Object = Object.create(null)
const prevChildren: Array<VNode> = this.prevChildren = this.children
const rawChildren: Array<VNode> = this.$slots.default || []
const children: Array<VNode> = this.children = []
const transitionData: Object = extractTransitionData(this)
```

不同于 `<transition>` 组件，`<transition-group>` 组件非抽象组件，它会渲染成一个真实元素，默认 `tag` 是 `span`。`prevChildren` 用来存储上一次的子节点；`children` 用来存储当前的子节点；`rawChildren` 表示 `<transtition-group>` 包裹的原始子节点；`transtdta` 是从 `<transtition-group>` 组件上提取出来的一些渲染数据，这点和 `<transition>` 组件的实现是一样的。

- 遍历 `rawChildren`，初始化 `children`

```
for (let i = 0; i < rawChildren.length; i++) {
  const c: VNode = rawChildren[i]
  if (c.tag) {
    if (c.key != null && String(c.key).indexOf('__vlist') !== 0) {
      children.push(c)
      map[c.key] = c
      ;(c.data || (c.data = {})).transition = transitionData
    } else if (process.env.NODE_ENV !== 'production') {
      const opts: ?VNodeComponentOptions = c.componentOptions
      const name: string = opts ? (opts.Ctor.options.name || opts.tag || '') : c.tag
      warn(`<transition-group> children must be keyed: <${name}>`)
    }
  }
}
```

其实就是对 `rawChildren` 遍历，拿到每个 `vnode`，然后会判断每个 `vnode` 是否设置了 `key`，这个是 `<transition-group>` 对列表元素的要求。然后把 `vnode` 添加到 `children` 中，然后把刚刚提取的过渡数据 `transitionData` 添加的 `vnode.data.transition` 中，这点很关键，只有这样才能实现列表中单个元素的过渡动画。

- 处理 `prevChildren`

```
if (prevChildren) {
  const kept: Array<VNode> = []
  const removed: Array<VNode> = []
  for (let i = 0; i < prevChildren.length; i++) {
    const c: VNode = prevChildren[i]
```

```

    c.data.transition = transitionData
    c.data.pos = c.elm.getBoundingClientRect()
    if (map[c.key]) {
      kept.push(c)
    } else {
      removed.push(c)
    }
  }
  this.kept = h(tag, null, kept)
  this.removed = removed
}

return h(tag, null, children)

```

当有 `prevChildren` 的时候，我们会对它做遍历，获取到每个 `vnode`，然后把 `transitionData` 赋值到 `vnode.data.transition`，这个是为了当它在 `enter` 和 `leave` 的钩子函数中有过渡动画，我们在上节介绍 `transition` 的实现中说过。接着又调用了原生 DOM 的 `getBoundingClientRect` 方法获取到原生 DOM 的位置信息，记录到 `vnode.data.pos` 中，然后判断一下 `vnode.key` 是否在 `map` 中，如果在则放入 `kept` 中，否则表示该节点已被删除，放入 `removed` 中，然后通过执行 `h(tag, null, kept)` 渲染后放入 `this.kept` 中，把 `removed` 用 `this.removed` 保存。最后整个 `render` 函数通过 `h(tag, null, children)` 生成渲染 `vnode`。

如果 `transition-group` 只实现了这个 `render` 函数，那么每次插入和删除的元素的缓动动画是可以实现的，在我们的例子中，当新增一个元素，它的插入的过渡动画是有的，但是剩余元素平移的过渡效果是出不来的，所以接下来我们来分析 `<transition-group>` 组件是如何实现剩余元素平移的过渡效果的。

move 过渡实现

其实我们在实现元素的插入和删除，无非就是操作数据，控制它们的添加和删除。比如我们新增数据的时候，会添加一条数据，除了重新执行 `render` 函数渲染新的节点外，还要触发 `updated` 钩子函数，接着我们就来分析 `updated` 钩子函数的实现。

- 判断子元素是否定义 `move` 相关样式

```

const children: Array<VNode> = this.prevChildren
const moveClass: string = this.moveClass || ((this.name || 'v') + '-move')
if (!children.length || !this.hasMove(children[0].elm, moveClass)) {
  return
}

hasMove (el: any, moveClass: string): boolean {
  /* istanbul ignore if */
  if (!hasTransition) {
    return false
  }
  /* istanbul ignore if */
  if (this._hasMove) {

```

```

    return this._hasMove
}

// Detect whether an element with the move class applied has
// CSS transitions. Since the element may be inside an entering
// transition at this very moment, we make a clone of it and remove
// all other transition classes applied to ensure only the move class
// is applied.
const clone: HTMLElement = el.cloneNode()
if (el._transitionClasses) {
  el._transitionClasses.forEach((cls: string) => { removeClass(clone, cls) })
}
addClass(clone, moveClass)
clone.style.display = 'none'
this.$el.appendChild(clone)
const info: Object = getTransitionInfo(clone)
this.$el.removeChild(clone)
return (this._hasMove = info.hasTransform)
}

```

核心就是 `hasMove` 的判断，首先克隆一个 DOM 节点，然后为了避免影响，移除它的所有其他的过渡 Class；接着添加了 `moveClass` 样式，设置 `display` 为 `none`，添加到组件根节点上；接下来通过 `getTransitionInfo` 获取它的一些缓动相关的信息，这个函数在上一节我们也介绍过，然后从组件根节点上删除这个克隆节点，并通过判断 `info.hasTransform` 来判断 `hasMove`，在我们的例子中，该值为 `true`。

- 子节点预处理

```

children.forEach(callPendingCbs)
children.forEach(recordPosition)
children.forEach(applyTranslation)

```

对 `children` 做了 3 轮循环，分别做了如下一些处理：

```

function callPendingCbs (c: VNode) {
  if (c.elm._moveCb) {
    c.elm._moveCb()
  }
  if (c.elm._enterCb) {
    c.elm._enterCb()
  }
}

function recordPosition (c: VNode) {
  c.data.newPos = c.elm.getBoundingClientRect()
}

function applyTranslation (c: VNode) {
  const oldPos = c.data.pos
  const newPos = c.data.newPos

```

```

const dx = oldPos.left - newPos.left
const dy = oldPos.top - newPos.top
if (dx || dy) {
  c.data.moved = true
  const s = c.elm.style
  s.transform = s.WebkitTransform = `translate(${dx}px, ${dy}px)`
  s.transitionDuration = '0s'
}
}

```

`callPendingCbs` 方法是在前一个过渡动画没执行完又再次执行到该方法的时候，会提前执行 `_moveCb` 和 `_enterCb`。

`recordPosition` 的作用是记录节点的新位置。

`applyTranslation` 的作用是先计算节点新位置和旧位置的差值，如果差值不为 0，则说明这些节点是需要移动的，所以记录 `vnode.data.moved` 为 true，并且通过设置 `transform` 把需要移动的节点的位置又偏移到之前的旧位置，目的是为了做 `move` 缓动做准备。

- 遍历子元素实现 move 过渡

```

this._reflow = document.body.offsetHeight

children.forEach((c: VNode) => {
  if (c.data.moved) {
    var el: any = c.elm
    var s: any = el.style
    addTransitionClass(el, moveClass)
    s.transform = s.WebkitTransform = s.transitionDuration = ''
    el.addEventListener(transitionEndEvent, el._moveCb = function cb (e) {
      if (!e || /transform$/.test(e.propertyName)) {
        el.removeEventListener(transitionEndEvent, cb)
        el._moveCb = null
        removeTransitionClass(el, moveClass)
      }
    })
  }
})

```

首先通过 `document.body.offsetHeight` 强制触发浏览器重绘，接着再次对 `children` 遍历，先给子节点添加 `moveClass`，在我们的例子中，`moveClass` 定义了 `transition: all 1s;` 缓动；接着把子节点的 `style.transform` 设置为空，由于我们前面把这些节点偏移到之前的旧位置，所以它就会从旧位置按照 `1s` 的缓动时间过渡偏移到它的当前目标位置，这样就实现了 `move` 的过渡动画。并且接下来会监听 `transitionEndEvent` 过渡结束的事件，做一些清理的操作。

另外，由于虚拟 DOM 的子元素更新算法是不稳定的，它不能保证被移除元素的相对位置，所以我们强制 `<transition-group>` 组件更新子节点通过 2 个步骤：第一步我们移除需要移除的 `vnode`，同时触发它们的 `leaving` 过渡；第二步我们需要把插入和移动的节点达到它们的最终态，同时还要保证移除的节点保留在应该的位置，而这个是通过 `beforeMount` 钩子函数来实现的：

```
beforeMount () {
  const update = this._update
  this._update = (vnode, hydrating) => {
    // force removing pass
    this.__patch__(
      this._vnode,
      this.kept,
      false, // hydrating
      true // removeOnly (!important, avoids unnecessary moves)
    )
    this._vnode = this.kept
    update.call(this, vnode, hydrating)
  }
}
```

通过把 `__patch__` 方法的第四个参数 `removeOnly` 设置为 `true`, 这样在 `updateChildren` 阶段, 是不会移动 `vnode` 节点的。

总结

那么到此, `<transtion-group>` 组件的实现原理就介绍完毕了, 它和 `<transition>` 组件相比, 实现了列表的过渡, 以及它会渲染成真实的元素。当我们去修改列表的数据的时候, 如果是添加或者删除数据, 则会触发相应元素本身的过渡动画, 这点和 `<transition>` 组件实现效果一样, 除此之外 `<transtion-group>` 还实现了 move 的过渡效果, 让我们的列表过渡动画更加丰富。

Vue-Router

路由的概念相信大部分同学并不陌生，它的作用就是根据不同的路径映射到不同的视图。我们在用 Vue 开发过实际项目的时候都会用到 Vue-Router 这个官方插件来帮我们解决路由的问题。Vue-Router 的能力十分强大，它支持 `hash`、`history`、`abstract` 3 种路由方式，提供了 `<router-link>` 和 `<router-view>` 2 种组件，还提供了简单的路由配置和一系列好用的 API。

大部分同学已经掌握了路由的基本使用，但使用的过程中也难免会遇到一些坑，那么这一章我们就来深挖 Vue-Router 的实现细节，一旦我们掌握了它的实现原理，那么就能在开发中会路由的使用更加游刃有余。

同样我们也会通过一些具体的示例来配合讲解，先来看一个最基本使用例子：

```
<div id="app">
  <h1>Hello App!</h1>
  <p>
    <!-- 使用 router-link 组件来导航. -->
    <!-- 通过传入 `to` 属性指定链接. -->
    <!-- <router-link> 默认会被渲染成一个 `<a>` 标签 -->
    <router-link to="/foo">Go to Foo</router-link>
    <router-link to="/bar">Go to Bar</router-link>
  </p>
  <!-- 路由出口 -->
  <!-- 路由匹配到的组件将渲染在这里 -->
  <router-view></router-view>
</div>
```

```
import Vue from 'vue'
import VueRouter from 'vue-router'

Vue.use(VueRouter)

// 1. 定义（路由）组件。
// 可以从其他文件 import 进来
const Foo = { template: '<div>foo</div>' }
const Bar = { template: '<div>bar</div>' }

// 2. 定义路由
// 每个路由应该映射一个组件。 其中"component" 可以是
// 通过 Vue.extend() 创建的组件构造器，
// 或者，只是一个组件配置对象。
// 我们晚点再讨论嵌套路由。
const routes = [
  { path: '/foo', component: Foo },
  { path: '/bar', component: Bar }
]
```

```
// 3. 创建 router 实例，然后传 `routes` 配置
// 你还可以传别的配置参数，不过先这么简单着吧。
const router = new VueRouter({
  routes // (缩写) 相当于 routes: routes
})

// 4. 创建和挂载根实例。
// 记得要通过 router 配置参数注入路由，
// 从而让整个应用都有路由功能
const app = new Vue({
  router
}).$mount('#app')
```

这是一个非常简单的例子，接下来我们先从 `Vue.use(VueRouter)` 说起。

路由注册

Vue 从它的设计上就是一个渐进式 JavaScript 框架，它本身的核心是解决视图渲染的问题，其它的能力就通过插件的方式来解决。Vue-Router 就是官方维护的路由插件，在介绍它的注册实现之前，我们先来分析一下 Vue 通用的插件注册原理。

Vue.use

Vue 提供了 `Vue.use` 的全局 API 来注册这些插件，所以我们先来分析一下它的实现原理，定义在 `vue/src/core/global-api/use.js` 中：

```
export function initUse (Vue: GlobalAPI) {
  Vue.use = function (plugin: Function | Object) {
    const installedPlugins = (this._installedPlugins || (this._installedPlugins = []))
    if (installedPlugins.indexOf(plugin) > -1) {
      return this
    }

    const args = toArray(arguments, 1)
    args.unshift(this)
    if (typeof plugin.install === 'function') {
      plugin.install.apply(plugin, args)
    } else if (typeof plugin === 'function') {
      plugin.apply(null, args)
    }
    installedPlugins.push(plugin)
    return this
  }
}
```

`Vue.use` 接受一个 `plugin` 参数，并且维护了一个 `_installedPlugins` 数组，它存储所有注册过的 `plugin`；接着又会判断 `plugin` 有没有定义 `install` 方法，如果说有的话则调用该方法，并且该方法执行的第一个参数是 `vue`；最后把 `plugin` 存储到 `installedPlugins` 中。

可以看到 Vue 提供的插件注册机制很简单，每个插件都需要实现一个静态的 `install` 方法，当我们执行 `Vue.use` 注册插件的时候，就会执行这个 `install` 方法，并且在这个 `install` 方法的第一个参数我们可以拿到 `Vue` 对象，这样好处就是作为插件的编写方不需要再额外去 `import Vue` 了。

路由安装

Vue-Router 的入口文件是 `src/index.js`，其中定义了 `VueRouter` 类，也实现了 `install` 的静态方法：`VueRouter.install = install`，它的定义在 `src/install.js` 中。

```

export let _Vue
export function install (Vue) {
  if (install.installed && _Vue === Vue) return
  install.installed = true

  _Vue = Vue

  const isDef = v => v !== undefined

  const registerInstance = (vm, callVal) => {
    let i = vm.$options._parentVnode
    if (isDef(i) && isDef(i = i.data) && isDef(i = i.registerRouteInstance)) {
      i(vm, callVal)
    }
  }

  Vue.mixin({
    beforeCreate () {
      if (isDef(this.$options.router)) {
        this._routerRoot = this
        this._router = this.$options.router
        this._router.init(this)
        Vue.util.defineReactive(this, '_route', this._router.history.current)
      } else {
        this._routerRoot = (this.$parent && this.$parent._routerRoot) || this
      }
      registerInstance(this, this)
    },
    destroyed () {
      registerInstance(this)
    }
  })

  Object.defineProperty(Vue.prototype, '$router', {
    get () { return this._routerRoot._router }
  })

  Object.defineProperty(Vue.prototype, '$route', {
    get () { return this._routerRoot._route }
  })

  Vue.component('RouterView', View)
  Vue.component('RouterLink', Link)

  const strats = Vue.config.optionMergeStrategies
  strats.beforeRouteEnter = strats.beforeRouteLeave = strats.beforeRouteUpdate = strats.created
}

```

当用户执行 `Vue.use(VueRouter)` 的时候，实际上就是在执行 `install` 函数，为了确保 `install` 逻辑只执行一次，用了 `install.installed` 变量做已安装的标志位。另外用一个全局的 `_Vue` 来接收参数 `vue`，因为作为 Vue 的插件对 `Vue` 对象是有依赖的，但又不能去单独去 `import Vue`，因为那样会增加包体积，所以就通过这种方式拿到 `Vue` 对象。

Vue-Router 安装最重要的一步就是利用 `Vue.mixin` 去把 `beforeCreate` 和 `destroyed` 钩子函数注入到每一个组件中。`Vue.mixin` 的定义，在 `vue/src/core/global-api/mixin.js` 中：

```
export function initMixin (Vue: GlobalAPI) {
  Vue.mixin = function (mixin: Object) {
    this.options = mergeOptions(this.options, mixin)
    return this
  }
}
```

它的实现实际上非常简单，就是把要混入的对象通过 `mergeOption` 合并到 `Vue` 的 `options` 中，由于每个组件的构造函数都会在 `extend` 阶段合并 `Vue.options` 到自身的 `options` 中，所以也就相当于每个组件都定义了 `mixin` 定义的选项。

回到 `Vue-Router` 的 `install` 方法，先看混入的 `beforeCreated` 钩子函数，对于根 `Vue` 实例而言，执行该钩子函数时定义了 `this._routerRoot` 表示它自身；`this._router` 表示 `VueRouter` 的实例 `router`，它是在 `new Vue` 的时候传入的；另外执行了 `this._router.init()` 方法初始化 `router`，这个逻辑之后介绍，然后用 `defineReactive` 方法把 `this._route` 变成响应式对象，这个作用我们之后会介绍。而对于子组件而言，由于组件是树状结构，在遍历组件树的过程中，它们在执行该钩子函数的时候 `this._routerRoot` 始终指向的是根 `Vue` 实例。

对于 `beforeCreated` 和 `destroyed` 钩子函数，它们都会执行 `registerInstance` 方法，这个方法的作用我们也是之后会介绍。

接着给 `Vue` 原型上定义了 `$router` 和 `$route` 2 个属性的 `get` 方法，这就是为什么我们可以在组件实例上可以访问 `this.$router` 以及 `this.$route`，它们的作用之后介绍。

接着又通过 `Vue.component` 方法定义了全局的 `<router-link>` 和 `<router-view>` 2 个组件，这也是为什么我们在写模板的时候可以使用这两个标签，它们的作用也是之后介绍。

最后定义了路由中的钩子函数的合并策略，和普通的钩子函数一样。

总结

那么到此为止，我们分析了 `Vue-Router` 的安装过程，`Vue` 编写插件的时候一定要提供静态的 `install` 方法，我们通过 `Vue.use(plugin)` 时候，就是在执行 `install` 方法。`Vue-Router` 的 `install` 方法会给每一个组件注入 `beforeCreated` 和 `destroyed` 钩子函数，在 `beforeCreated` 做一些私有属性定义和路由初始化工作，下一节我们就来分析一下 `VueRouter` 对象的实现和它的初始化工作。

VueRouter 对象

VueRouter 的实现是一个类，我们先对它做一个简单地分析，它的定义在 `src/index.js` 中：

```

export default class Router {
  static install: () => void;
  static version: string;

  app: any;
  apps: Array<any>;
  ready: boolean;
  readyCbs: Array<Function>;
  options: RouterOptions;
  mode: string;
  history: HashHistory | HTML5History | AbstractHistory;
  matcher: Matcher;
  fallback: boolean;
  beforeHooks: Array<?NavigationGuard>;
  resolveHooks: Array<?NavigationGuard>;
  afterHooks: Array<?AfterNavigationHook>;

  constructor (options: RouterOptions = {}) {
    this.app = null
    this.apps = []
    this.options = options
    this.beforeHooks = []
    this.resolveHooks = []
    this.afterHooks = []
    this.matcher = createMatcher(options.routes || [], this)

    let mode = options.mode || 'hash'
    this.fallback = mode === 'history' && !supportsPushState && options.fallback != false
    if (this.fallback) {
      mode = 'hash'
    }
    if (!inBrowser) {
      mode = 'abstract'
    }
    this.mode = mode

    switch (mode) {
      case 'history':
        this.history = new HTML5History(this, options.base)
        break
      case 'hash':
        this.history = new HashHistory(this, options.base, this.fallback)
        break
    }
  }
}

```

```

        case 'abstract':
            this.history = new AbstractHistory(this, options.base)
            break
        default:
            if (process.env.NODE_ENV !== 'production') {
                assert(false, `invalid mode: ${mode}`)
            }
        }
    }

match (
    raw: RawLocation,
    current?: Route,
    redirectedFrom?: Location
): Route {
    return this.matcher.match(raw, current, redirectedFrom)
}

get currentRoute (): ?Route {
    return this.history && this.history.current
}

init (app: any) {
    process.env.NODE_ENV !== 'production' && assert(
        install.installed,
        `not installed. Make sure to call \`Vue.use(VueRouter)\` +` +
        `before creating root instance.`
    )
    this.apps.push(app)

    if (this.app) {
        return
    }

    this.app = app

    const history = this.history

    if (history instanceof HTML5History) {
        history.transitionTo(history.getCurrentLocation())
    } else if (history instanceof HashHistory) {
        const setupHashListener = () => {
            history.setupListeners()
        }
        history.transitionTo(
            history.getCurrentLocation(),
            setupHashListener,
            setupHashListener
        )
    }
}

```

```

    history.listen(route => {
      this.apps.forEach((app) => {
        app._route = route
      })
    })
  }

  beforeEach (fn: Function): Function {
    return registerHook(this.beforeHooks, fn)
  }

  beforeResolve (fn: Function): Function {
    return registerHook(this.resolveHooks, fn)
  }

  afterEach (fn: Function): Function {
    return registerHook(this.afterHooks, fn)
  }

  onReady (cb: Function, errorCallback?: Function) {
    this.history.onReady(cb, errorCallback)
  }

  onError (errorCallback: Function) {
    this.history.onError(errorCallback)
  }

  push (location: RawLocation, onComplete?: Function, onAbort?: Function) {
    this.history.push(location, onComplete, onAbort)
  }

  replace (location: RawLocation, onComplete?: Function, onAbort?: Function) {
    this.history.replace(location, onComplete, onAbort)
  }

  go (n: number) {
    this.history.go(n)
  }

  back () {
    this.go(-1)
  }

  forward () {
    this.go(1)
  }

  getMatchedComponents (to?: RawLocation | Route): Array<any> {
    const route: any = to
    ? to.matched
  }

```

```

    ? to
      : this.resolve(to).route
    : this.currentRoute
  if (!route) {
    return []
  }
  return [].concat.apply([], route.matched.map(m => {
    return Object.keys(m.components).map(key => {
      return m.components[key]
    })
  }))
}
}

resolve (
  to: RawLocation,
  current?: Route,
  append?: boolean
): {
  location: Location,
  route: Route,
  href: string,
  normalizedTo: Location,
  resolved: Route
} {
  const location = normalizeLocation(
    to,
    current || this.history.current,
    append,
    this
  )
  const route = this.match(location, current)
  const fullPath = route.redirectedFrom || route.fullPath
  const base = this.history.base
  const href = createHref(base, fullPath, this.mode)
  return {
    location,
    route,
    href,
    normalizedTo: location,
    resolved: route
  }
}

addRoutes (routes: Array<RouteConfig>) {
  this.matcher.addRoutes(routes)
  if (this.history.current !== START) {
    this.history.transitionTo(this.history.getCurrentLocation())
  }
}
}

```

`VueRouter` 定义了一些属性和方法，我们先从它的构造函数看，当我们执行 `new VueRouter` 的时候做了哪些事情。

```

constructor (options: RouterOptions = {}) {
  this.app = null
  this.apps = []
  this.options = options
  this.beforeHooks = []
  this.resolveHooks = []
  this.afterHooks = []
  this.matcher = createMatcher(options.routes || [], this)

  let mode = options.mode || 'hash'
  this.fallback = mode === 'history' && !supportsPushState && options.fallback !==
false
  if (this.fallback) {
    mode = 'hash'
  }
  if (!inBrowser) {
    mode = 'abstract'
  }
  this.mode = mode

  switch (mode) {
    case 'history':
      this.history = new HTML5History(this, options.base)
      break
    case 'hash':
      this.history = new HashHistory(this, options.base, this.fallback)
      break
    case 'abstract':
      this.history = new AbstractHistory(this, options.base)
      break
    default:
      if (process.env.NODE_ENV !== 'production') {
        assert(false, `invalid mode: ${mode}`)
      }
  }
}

```

构造函数定义了一些属性，其中 `this.app` 表示根 `Vue` 实例，`this.apps` 保存所有子组件的 `Vue` 实例，`this.options` 保存传入的路由配置，`this.beforeHooks`、`this.resolveHooks`、`this.afterHooks` 表示一些钩子函数，我们之后会介绍，`this.matcher` 表示路由匹配器，我们之后会介绍，`this.fallback` 表示路由创建失败的回调函数，`this.mode` 表示路由创建的模式，`this.history` 表示路由历史的具体的实现实例，它是根据 `this.mode` 的不同实现不同，它有 `History` 基类，然后不同的 `history` 实现都是继承 `History`，这块我们之后会重点讲。

实例化 `VueRouter` 后会返回它的实例 `router`，我们在 `new Vue` 的时候会把 `router` 作为配置的属性传入，回顾一下上一节我们讲 `beforeCreated` 混入的时候有这么一段代码：

```
beforeCreated() {
  if (isDef(this.$options.router)) {
    // ...
    this._router = this.$options.router
    this._router.init(this)
    // ...
  }
}
```

所以每个组件在执行 `beforeCreated` 钩子函数的时候，都会执行 `router.init` 方法：

```
init (app: any) {
  process.env.NODE_ENV !== 'production' && assert(
    install.installed,
    `not installed. Make sure to call \`Vue.use(VueRouter)\` +` +
    `before creating root instance.`
  )

  this.apps.push(app)

  if (this.app) {
    return
  }

  this.app = app

  const history = this.history

  if (history instanceof HTML5History) {
    history.transitionTo(history.getCurrentLocation())
  } else if (history instanceof HashHistory) {
    const setupHashListener = () => {
      history.setupListeners()
    }
    history.transitionTo(
      history.getCurrentLocation(),
      setupHashListener,
      setupHashListener
    )
  }

  history.listen(route => {
    this.apps.forEach((app) => {
      app._route = route
    })
  })
}
```

```
}
```

`init` 的逻辑很简单，它传入的参数是 `Vue` 实例，然后存储到 `this.apps` 中；只有根 `Vue` 实例会保存到 `this.app` 中，并且会拿到当前的 `this.history`，根据它的不同类型来执行不同逻辑，由于我们平时使用 `hash` 路由多一些，所以我们先看这部分逻辑，先定义了 `setupHashListener` 函数，接着执行了 `history.transitionTo` 方法，它是定义在 `History` 基类中，代码在 `src/history/base.js`：

```
transitionTo (location: RawLocation, onComplete?: Function, onAbort?: Function) {
  const route = this.router.match(location, this.current)
  // ...
}
```

我们先不着急去看 `transitionTo` 的具体实现，先看第一行代码，它调用了 `this.router.match` 函数：

```
match (
  raw: RawLocation,
  current?: Route,
  redirectedFrom?: Location
): Route {
  return this.matcher.match(raw, current, redirectedFrom)
}
```

实际上是调用了 `this.matcher.match` 方法去做匹配，所以接下来我们先来了解一下 `matcher` 的相关实现。

总结

通过这一节的分析，我们大致对 `VueRouter` 类有了大致了解，知道了它的一些属性和方法，同时了解到在组件的初始化阶段，执行到 `beforeCreated` 钩子函数的时候会执行 `router.init` 方法，然后又会执行 `history.transitionTo` 方法做路由过渡，进而引出了 `matcher` 的概念，接下来我们先研究一下 `matcher` 的相关实现。

matcher

`matcher` 相关的实现都在 `src/create-matcher.js` 中，我们先来看一下 `matcher` 的数据结构：

```
export type Matcher = {
  match: (raw: RawLocation, current?: Route, redirectedFrom?: Location) => Route;
  addRoutes: (routes: Array<RouteConfig>) => void;
};
```

`Matcher` 返回了 2 个方法，`match` 和 `addRoutes`，在上一节我们接触到了 `match` 方法，顾名思义它是做匹配，那么匹配的是什么，在介绍之前，我们先了解路由中重要的 2 个概念，`Location` 和 `Route`，它们的数据结构定义在 `flow/declarations.js` 中。

- Location

```
declare type Location = {
  _normalized?: boolean;
  name?: string;
  path?: string;
  hash?: string;
  query?: Dictionary<string>;
  params?: Dictionary<string>;
  append?: boolean;
  replace?: boolean;
}
```

Vue-Router 中定义的 `Location` 数据结构和浏览器提供的 `window.location` 部分结构有点类似，它们都是对 `url` 的结构化描述。举个例子：`/abc?foo=bar&baz=qux#hello`，它的 `path` 是 `/abc`，`query` 是 `{foo:bar,baz:qux}`。`Location` 的其他属性我们之后会介绍。

- Route

```
declare type Route = {
  path: string;
  name: ?string;
  hash: string;
  query: Dictionary<string>;
  params: Dictionary<string>;
  fullPath: string;
  matched: Array<RouteRecord>;
  redirectedFrom?: string;
  meta?: any;
}
```

`Route` 表示的是路由中的一条线路，它除了描述了类似 `Location` 的 `path`、`query`、`hash` 这些概念，还有 `matched` 表示匹配到的所有的 `RouteRecord`。`Route` 的其他属性我们之后会介绍。

createMatcher

在了解了 `Location` 和 `Route` 后，我们来看一下 `matcher` 的创建过程：

```
export function createMatcher (
  routes: Array<RouteConfig>,
  router: VueRouter
): Matcher {
  const { pathList, pathMap, nameMap } = createRouteMap(routes)

  function addRoutes (routes) {
    createRouteMap(routes, pathList, pathMap, nameMap)
  }

  function match (
    raw: RawLocation,
    currentRoute?: Route,
    redirectedFrom?: Location
  ): Route {
    const location = normalizeLocation(raw, currentRoute, false, router)
    const { name } = location

    if (name) {
      const record = nameMap[name]
      if (process.env.NODE_ENV !== 'production') {
        warn(record, `Route with name '${name}' does not exist`)
      }
      if (!record) return _createRoute(null, location)
      const paramNames = record.regex.keys
        .filter(key => !key.optional)
        .map(key => key.name)

      if (typeof location.params !== 'object') {
        location.params = {}
      }

      if (currentRoute && typeof currentRoute.params === 'object') {
        for (const key in currentRoute.params) {
          if (!(key in location.params) && paramNames.indexOf(key) > -1) {
            location.params[key] = currentRoute.params[key]
          }
        }
      }
    }

    if (record) {
```

```

        location.path = fillParams(record.path, location.params, `named route "${name}`)
        return _createRoute(record, location, redirectedFrom)
    }
} else if (location.path) {
    location.params = {}
    for (let i = 0; i < pathList.length; i++) {
        const path = pathList[i]
        const record = pathMap[path]
        if (matchRoute(record.regex, location.path, location.params)) {
            return _createRoute(record, location, redirectedFrom)
        }
    }
}
return _createRoute(null, location)
}

// ...

function _createRoute (
    record: ?RouteRecord,
    location: Location,
    redirectedFrom?: Location
): Route {
    if (record && record.redirect) {
        return redirect(record, redirectedFrom || location)
    }
    if (record && record.matchAs) {
        return alias(record, location, record.matchAs)
    }
    return createRoute(record, location, redirectedFrom, router)
}

return {
    match,
    addRoutes
}
}

```

`createMatcher` 接收 2 个参数，一个是 `router`，它是我们 `new VueRouter` 返回的实例，一个是 `routes`，它是用户定义的路由配置，来看一下我们之前举的例子中的配置：

```

const Foo = { template: '<div>foo</div>' }
const Bar = { template: '<div>bar</div>' }

const routes = [
    { path: '/foo', component: Foo },
    { path: '/bar', component: Bar }
]

```

`createMatcher` 首先执行的逻辑是 `const { pathList, pathMap, nameMap } = createRouteMap(routes)` 创建一个路由映射表，`createRouteMap` 的定义在 `src/create-route-map` 中：

```
export function createRouteMap (
  routes: Array<RouteConfig>,
  oldPathList?: Array<string>,
  oldPathMap?: Dictionary<RouteRecord>,
  oldNameMap?: Dictionary<RouteRecord>
): {
  pathList: Array<string>;
  pathMap: Dictionary<RouteRecord>;
  nameMap: Dictionary<RouteRecord>;
} {
  const pathList: Array<string> = oldPathList || []
  const pathMap: Dictionary<RouteRecord> = oldPathMap || Object.create(null)
  const nameMap: Dictionary<RouteRecord> = oldNameMap || Object.create(null)

  routes.forEach(route => {
    addRouteRecord(pathList, pathMap, nameMap, route)
  })

  for (let i = 0, l = pathList.length; i < l; i++) {
    if (pathList[i] === '*') {
      pathList.push(pathList.splice(i, 1)[0])
      l--
      i--
    }
  }
}

return {
  pathList,
  pathMap,
  nameMap
}
}
```

`createRouteMap` 函数的目标是把用户的路由配置转换成一张路由映射表，它包含 3 个部分，`pathList` 存储所有的 `path`，`pathMap` 表示一个 `path` 到 `RouteRecord` 的映射关系，而 `nameMap` 表示 `name` 到 `RouteRecord` 的映射关系。那么 `RouteRecord` 到底是什么，先来看一下它的数据结构：

```
declare type RouteRecord = {
  path: string;
  regex: RouteRegExp;
  components: Dictionary<any>;
  instances: Dictionary<any>;
  name: ?string;
  parent: ?RouteRecord;
```

```

    redirect: ?RedirectOption;
    matchAs: ?string;
    beforeEnter: ?NavigationGuard;
    meta: any;
    props: boolean | Object | Function | Dictionary<boolean | Object | Function>;
}

```

它的创建是通过遍历 `routes` 为每一个 `route` 执行 `addRouteRecord` 方法生成一条记录，来看一下它的定义：

```

function addRouteRecord (
  pathList: Array<string>,
  pathMap: Dictionary<RouteRecord>,
  nameMap: Dictionary<RouteRecord>,
  route: RouteConfig,
  parent?: RouteRecord,
  matchAs?: string
) {
  const { path, name } = route
  if (process.env.NODE_ENV !== 'production') {
    assert(path != null, `"path" is required in a route configuration.`)
    assert(
      typeof route.component !== 'string',
      `route config "component" for path: ${String(path || name)} cannot be a ` +
      `string id. Use an actual component instead.`
    )
  }

  const pathToRegexpOptions: PathToRegexpOptions = route.pathToRegexpOptions || {}
  const normalizedPath = normalizePath(
    path,
    parent,
    pathToRegexpOptions.strict
  )

  if (typeof route.caseSensitive === 'boolean') {
    pathToRegexpOptions.sensitive = route.caseSensitive
  }

  const record: RouteRecord = {
    path: normalizedPath,
    regex: compileRouteRegex(normalizedPath, pathToRegexpOptions),
    components: route.components || { default: route.component },
    instances: {},
    name,
    parent,
    matchAs,
    redirect: route.redirect,
    beforeEnter: route.beforeEnter,
    meta: route.meta || {}
  }
}

```

```

props: route.props == null
? {}
: route.components
? route.props
: { default: route.props }
}

if (route.children) {
  if (process.env.NODE_ENV !== 'production') {
    if (route.name && !route.redirect && route.children.some(child => /^\/?$/ .test(child.path))) {
      warn(
        false,
        `Named Route '${route.name}' has a default child route. ` +
        `When navigating to this named route (:to="{name: '${route.name}'})", ` +
        `the default child route will not be rendered. Remove the name from ` +
        `this route and use the name of the default child route for named ` +
        `links instead.`
      )
    }
  }
  route.children.forEach(child => {
    const childMatchAs = matchAs
    ? cleanPath(`.${matchAs}/${child.path}`)
    : undefined
    addRouteRecord(pathList, pathMap, nameMap, child, record, childMatchAs)
  })
}

if (route.alias !== undefined) {
  const aliases = Array.isArray(route.alias)
  ? route.alias
  : [route.alias]

  aliases.forEach(alias => {
    const aliasRoute = {
      path: alias,
      children: route.children
    }
    addRouteRecord(
      pathList,
      pathMap,
      nameMap,
      aliasRoute,
      parent,
      record.path || '/'
    )
  })
}

if (!pathMap[record.path]) {

```

```

    pathList.push(record.path)
    pathMap[record.path] = record
}

if (name) {
  if (!nameMap[name]) {
    nameMap[name] = record
  } else if (process.env.NODE_ENV !== 'production' && !matchAs) {
    warn(
      false,
      `Duplicate named routes definition: ` +
      `{ name: "${name}", path: "${record.path}" }` )
  }
}
}
}

```

我们只看几个关键逻辑，首先创建 `RouteRecord` 的代码如下：

```

const record: RouteRecord = {
  path: normalizedPath,
  regex: compileRouteRegex(normalizedPath, pathToRegexpOptions),
  components: route.components || { default: route.component },
  instances: {},
  name,
  parent,
  matchAs,
  redirect: route.redirect,
  beforeEnter: route.beforeEnter,
  meta: route.meta || {},
  props: route.props == null
    ? {}
    : route.components
      ? route.props
      : { default: route.props }
}

```

这里要注意几个点，`path` 是规范化后的路径，它会根据 `parent` 的 `path` 做计算；`regex` 是一个正则表达式的扩展，它利用了 `path-to-regexp` 这个工具库，把 `path` 解析成一个正则表达式的扩展，举个例子：

```

var keys = []
var re = pathToRegexp('/foo/:bar', keys)
// re = /^\/foo\/([^\/]++)\/?$/i
// keys = [{ name: 'bar', prefix: '/', delimiter: '/', optional: false, repeat: false, pattern: '[^\\/]++' }]

```

`components` 是一个对象，通常我们在配置中写的 `component` 实际上这里会被转换成 `{components: route.component}`；`instances` 表示组件的实例，也是一个对象类型；`parent` 表示父的 `RouteRecord`，因为我们配置的时候有时候会配置子路由，所以整个 `RouteRecord` 也就是一个树型结构。

```
if (route.children) {
  // ...
  route.children.forEach(child => {
    const childMatchAs = matchAs
    ? cleanPath(` ${matchAs}/${child.path}`)
    : undefined
    addRouteRecord(pathList, pathMap, nameMap, child, record, childMatchAs)
  })
}
```

如果配置了 `children`，那么递归执行 `addRouteRecord` 方法，并把当前的 `record` 作为 `parent` 传入，通过这样的深度遍历，我们就可以拿到一个 `route` 下的完整记录。

```
if (!pathMap[record.path]) {
  pathList.push(record.path)
  pathMap[record.path] = record
}
```

为 `pathList` 和 `pathMap` 各添加一条记录。

```
if (name) {
  if (!nameMap[name]) {
    nameMap[name] = record
  }
  // ...
}
```

如果我们在路由配置中配置了 `name`，则给 `nameMap` 添加一条记录。

由于 `pathList`、`pathMap`、`nameMap` 都是引用类型，所以在遍历整个 `routes` 过程中去执行 `addRouteRecord` 方法，会不断给他们添加数据。那么经过整个 `createRouteMap` 方法的执行，我们得到的就是 `pathList`、`pathMap` 和 `nameMap`。其中 `pathList` 是为了记录路由配置中的所有 `path`，而 `pathMap` 和 `nameMap` 都是为了通过 `path` 和 `name` 能快速查到对应的 `RouteRecord`。

再回到 `createMatcher` 函数，接下来就定义了一系列方法，最后返回了一个对象。

```
return {
  match,
  addRoutes
}
```

也就是说，`matcher` 是一个对象，它对外暴露了 `match` 和 `addRoutes` 方法。

addRoutes

`addRoutes` 方法的作用是动态添加路由配置，因为在实际开发中有些场景是不能提前把路由写死的，需要根据一些条件动态添加路由，所以 Vue-Router 也提供了这一接口：

```
function addRoutes (routes) {
  createRouteMap(routes, pathList, pathMap, nameMap)
}
```

`addRoutes` 的方法十分简单，再次调用 `createRouteMap` 即可，传入新的 `routes` 配置，由于 `pathList`、`pathMap`、`nameMap` 都是引用类型，执行 `addRoutes` 后会修改它们的值。

match

```
function match (
  raw: RawLocation,
  currentRoute?: Route,
  redirectedFrom?: Location
): Route {
  const location = normalizeLocation(raw, currentRoute, false, router)
  const { name } = location

  if (name) {
    const record = nameMap[name]
    if (process.env.NODE_ENV !== 'production') {
      warn(record, `Route with name '${name}' does not exist`)
    }
    if (!record) return _createRoute(null, location)
    const paramNames = record.regex.keys
      .filter(key => !key.optional)
      .map(key => key.name)

    if (typeof location.params !== 'object') {
      location.params = {}
    }

    if (currentRoute && typeof currentRoute.params === 'object') {
      for (const key in currentRoute.params) {
        if (!(key in location.params) && paramNames.indexOf(key) > -1) {
          location.params[key] = currentRoute.params[key]
        }
      }
    }
  }

  if (record) {
```

```

        location.path = fillParams(record.path, location.params, `named route "${name}`
```)
 return _createRoute(record, location, redirectedFrom)
 }
} else if (location.path) {
 location.params = {}
 for (let i = 0; i < pathList.length; i++) {
 const path = pathList[i]
 const record = pathMap[path]
 if (matchRoute(record.regex, location.path, location.params)) {
 return _createRoute(record, location, redirectedFrom)
 }
 }
}

return _createRoute(null, location)
}

```

`match` 方法接收 3 个参数，其中 `raw` 是 `RawLocation` 类型，它可以是一个 `url` 字符串，也可以是一个 `Location` 对象；`currentRoute` 是 `Route` 类型，它表示当前的路径；`redirectedFrom` 和重定向相关，这里先忽略。`match` 方法返回的是一个路径，它的作用是根据传入的 `raw` 和当前的路径 `currentRoute` 计算出一个新的路径并返回。

首先执行了 `normalizeLocation`，它的定义在 `src/util/location.js` 中：

```

export function normalizeLocation (
 raw: RawLocation,
 current: ?Route,
 append: ?boolean,
 router: ?VueRouter
): Location {
 let next: Location = typeof raw === 'string' ? { path: raw } : raw
 if (next.name || next._normalized) {
 return next
 }

 if (!next.path && next.params && current) {
 next = assign({}, next)
 next._normalized = true
 const params: any = assign(assign({}, current.params), next.params)
 if (current.name) {
 next.name = current.name
 next.params = params
 } else if (current.matched.length) {
 const rawPath = current.matched[current.matched.length - 1].path
 next.path = fillParams(rawPath, params, `path ${current.path}`)
 } else if (process.env.NODE_ENV !== 'production') {
 warn(false, `relative params navigation requires a current route.`)
 }
 }
}

```

```

 return next
 }

 const parsedPath = parsePath(next.path || '/')
 const basePath = (current && current.path) || '/'
 const path = parsedPath.path
 ? resolvePath(parsedPath.path, basePath, append || next.append)
 : basePath

 const query = resolveQuery(
 parsedPath.query,
 next.query,
 router && router.options.parseQuery
)

 let hash = next.hash || parsedPath.hash
 if (hash && hash.charAt(0) !== '#') {
 hash = `#${hash}`
 }

 return {
 _normalized: true,
 path,
 query,
 hash
 }
}

```

`normalizeLocation` 方法的作用是根据 `raw`，`current` 计算出新的 `location`，它主要处理了 `raw` 的两种情况，一种是有 `params` 且没有 `path`，一种是有 `path` 的，对于第一种情况，如果 `current` 有 `name`，则计算出的 `location` 也有 `name`。

计算出新的 `location` 后，对 `location` 的 `name` 和 `path` 的两种情况做了处理。

- `name`

有 `name` 的情况下就根据 `nameMap` 匹配到 `record`，它就是一个 `RouterRecord` 对象，如果 `record` 不存在，则匹配失败，返回一个空路径；然后拿到 `record` 对应的 `paramNames`，再对比 `currentRoute` 中的 `params`，把交集部分的 `params` 添加到 `location` 中，然后在通过 `fillParams` 方法根据 `record.path` 和 `location.path` 计算出 `location.path`，最后调用 `_createRoute(record, location, redirectedFrom)` 去生成一条新路径，该方法我们之后会介绍。

- `path`

通过 `name` 我们可以很快的找到 `record`，但是通过 `path` 并不能，因为我们计算后的 `location.path` 是一个真实路径，而 `record` 中的 `path` 可能会有 `param`，因此需要对所有的 `pathList` 做顺序遍历，然后通过 `matchRoute` 方法根据

`record.regex`、`location.path`、`location.params` 匹配，如果匹配到则也通过 `_createRoute(record, location, redirectedFrom)` 去生成一条新路径。因为是顺序遍历，所以我们书写路由配置要注意路径的顺序，因为写在前面的会优先尝试匹配。

最后我们来看一下 `_createRoute` 的实现：

```
function _createRoute (
 record: ?RouteRecord,
 location: Location,
 redirectedFrom?: Location
): Route {
 if (record && record.redirect) {
 return redirect(record, redirectedFrom || location)
 }
 if (record && record.matchAs) {
 return alias(record, location, record.matchAs)
 }
 return createRoute(record, location, redirectedFrom, router)
}
```

我们先不考虑 `record.redirect` 和 `record.matchAs` 的情况，最终会调用 `createRoute` 方法，它的定义在 `src/util/route.js` 中：

```
export function createRoute (
 record: ?RouteRecord,
 location: Location,
 redirectedFrom?: ?Location,
 router?: VueRouter
): Route {
 const stringifyQuery = router && router.options.stringifyQuery

 let query: any = location.query || {}
 try {
 query = clone(query)
 } catch (e) {}

 const route: Route = {
 name: location.name || (record && record.name),
 meta: (record && record.meta) || {},
 path: location.path || '/',
 hash: location.hash || '',
 query,
 params: location.params || {},
 fullPath: getFullPath(location, stringifyQuery),
 matched: record ? formatMatch(record) : []
 }
 if (redirectedFrom) {
 route.redirectedFrom = getFullPath(redirectedFrom, stringifyQuery)
 }
}
```

```
 return Object.freeze(route)
}
```

`createRoute` 可以根据 `record` 和 `location` 创建出来，最终返回的是一条 `Route` 路径，我们之前也介绍过它的数据结构。在 Vue-Router 中，所有的 `Route` 最终都会通过 `createRoute` 函数创建，并且它最后是不可以被外部修改的。`Route` 对象中有一个非常重要属性是 `matched`，它通过 `formatMatch(record)` 计算而来：

```
function formatMatch (record: ?RouteRecord): Array<RouteRecord> {
 const res = []
 while (record) {
 res.unshift(record)
 record = record.parent
 }
 return res
}
```

可以看它是通过 `record` 循环向上找 `parent`，只到找到最外层，并把所有的 `record` 都 push 到一个数组中，最终返回的就是 `record` 的数组，它记录了一条线路上的所有 `record`。`matched` 属性非常有用，它为之后渲染组件提供了依据。

## 总结

那么到此，`matcher` 相关的主流程的分析就结束了，我们了解了 `Location`、`Route`、`RouteRecord` 等概念。并通过 `matcher` 的 `match` 方法，我们会找到匹配的路径 `Route`，这个对 `Route` 的切换，组件的渲染都有非常重要的指导意义。下一节我们会回到 `transitionTo` 方法，看一看路径的切换都做了哪些事情。

## 路径切换

`history.transitionTo` 是 Vue-Router 中非常重要的方法，当我们切换路由线路的时候，就会执行到该方法，前一节我们分析了 `matcher` 的相关实现，知道它是如何找到匹配的新线路，那么匹配到新线路后又做了哪些事情，接下来我们来完整分析一下 `transitionTo` 的实现，它的定义在

`src/history/base.js` 中：

```
transitionTo (location: RawLocation, onComplete?: Function, onAbort?: Function) {
 const route = this.router.match(location, this.current)
 this.confirmTransition(route, () => {
 this.updateRoute(route)
 onComplete && onComplete(route)
 this.ensureURL()

 if (!this.ready) {
 this.ready = true
 this.readyCbs.forEach(cb => { cb(route) })
 }
 }, err => {
 if (onAbort) {
 onAbort(err)
 }
 if (err && !this.ready) {
 this.ready = true
 this.readyErrorCbs.forEach(cb => { cb(err) })
 }
 })
}
```

`transitionTo` 首先根据目标 `location` 和当前路径 `this.current` 执行 `this.router.match` 方法去匹配到目标的路径。这里 `this.current` 是 `history` 维护的当前路径，它的初始值是在 `history` 的构造函数中初始化的：

```
this.current = START
```

`START` 的定义在 `src/util/route.js` 中：

```
export const START = createRoute(null, {
 path: '/'
})
```

这样就创建了一个初始的 `Route`，而 `transitionTo` 实际上也就是在切换 `this.current`，稍后我们会看到。

拿到新的路径后，那么接下来就会执行 `confirmTransition` 方法去做真正的切换，由于这个过程可能有一些异步的操作（如异步组件），所以整个 `confirmTransition` API 设计成带有成功回调函数和失败回调函数，先来看一下它的定义：

```
confirmTransition (route: Route, onComplete: Function, onAbort?: Function) {
 const current = this.current
 const abort = err => {
 if (isError(err)) {
 if (this.errorCbs.length) {
 this.errorCbs.forEach(cb => { cb(err) })
 } else {
 warn(false, 'uncaught error during route navigation:')
 console.error(err)
 }
 }
 onAbort && onAbort(err)
 }
 if (
 isSameRoute(route, current) &&
 route.matched.length === current.matched.length
) {
 this.ensureURL()
 return abort()
 }

 const {
 updated,
 deactivated,
 activated
 } = resolveQueue(this.current.matched, route.matched)

 const queue: Array<?NavigationGuard> = [].concat(
 extractLeaveGuards(deactivated),
 this.router.beforeHooks,
 extractUpdateHooks(updated),
 activated.map(m => m.beforeEnter),
 resolveAsyncComponents(activated)
)

 this.pending = route
 const iterator = (hook: NavigationGuard, next) => {
 if (this.pending !== route) {
 return abort()
 }
 try {
 hook(route, current, (to: any) => {
 if (to === false || isError(to)) {
 this.ensureURL(true)
 abort(to)
 } else if (

```

```

 typeof to === 'string' ||
 (typeof to === 'object' && (
 typeof to.path === 'string' ||
 typeof to.name === 'string'
))
) {
 abort()
 if (typeof to === 'object' && to.replace) {
 this.replace(to)
 } else {
 this.push(to)
 }
 } else {
 next(to)
 }
}
} catch (e) {
 abort(e)
}
}

runQueue(queue, iterator, () => {
 const postEnterCbs = []
 const isValid = () => this.current === route
 const enterGuards = extractEnterGuards(activated, postEnterCbs, isValid)
 const queue = enterGuards.concat(this.router.resolveHooks)
 runQueue(queue, iterator, () => {
 if (this.pending !== route) {
 return abort()
 }
 this.pending = null
 onComplete(route)
 if (this.router.app) {
 this.router.app.$nextTick(() => {
 postEnterCbs.forEach(cb => { cb() })
 })
 }
 })
})
}
}

```

首先定义了 `abort` 函数，然后判断如果满足计算后的 `route` 和 `current` 是相同路径的话，则直接调用 `this.ensureUrl` 和 `abort`，`ensureUrl` 这个函数我们之后会介绍。

接着又根据 `current.matched` 和 `route.matched` 执行了 `resolveQueue` 方法解析出 3 个队列：

```

function resolveQueue (
 current: Array<RouteRecord>,
 next: Array<RouteRecord>
): {

```

```

updated: Array<RouteRecord>,
activated: Array<RouteRecord>,
deactivated: Array<RouteRecord>
} {
let i
const max = Math.max(current.length, next.length)
for (i = 0; i < max; i++) {
 if (current[i] !== next[i]) {
 break
 }
}
return {
 updated: next.slice(0, i),
 activated: next.slice(i),
 deactivated: current.slice(i)
}
}
}

```

因为 `route.matched` 是一个 `RouteRecord` 的数组，由于路径是由 `current` 变向 `route`，那么就遍历对比 2 边的 `RouteRecord`，找到一个不一样的位置 `i`，那么 `next` 中从 0 到 `i` 的 `RouteRecord` 是两边都一样，则为 `updated` 的部分；从 `i` 到最后的 `RouteRecord` 是 `next` 独有的，为 `activated` 的部分；而 `current` 中从 `i` 到最后的 `RouteRecord` 则没有了，为 `deactivated` 的部分。

拿到 `updated`、`activated`、`deactivated` 3 个 `ReouteRecord` 数组后，接下来就是路径变换后的一个重要部分，执行一系列的钩子函数。

## 导航守卫

官方的说法叫导航守卫，实际上就是发生在路由路径切换的时候，执行的一系列钩子函数。

我们先从整体上看一下这些钩子函数执行的逻辑，首先构造一个队列 `queue`，它实际上是一个数组；然后再定义一个迭代器函数 `iterator`；最后再执行 `runQueue` 方法来执行这个队列。我们先来看一下 `runQueue` 的定义，在 `src/util/async.js` 中：

```

export function runQueue (queue: Array<?NavigationGuard>, fn: Function, cb: Function
) {
 const step = index => {
 if (index >= queue.length) {
 cb()
 } else {
 if (queue[index]) {
 fn(queue[index], () => {
 step(index + 1)
 })
 } else {
 step(index + 1)
 }
 }
 }
}

```

```

 }
 }
 step(0)
}

```

这是一个非常经典的异步函数队列化执行的模式，`queue` 是一个 `NavigationGuard` 类型的数组，我们定义了 `step` 函数，每次根据 `index` 从 `queue` 中取一个 `guard`，然后执行 `fn` 函数，并且把 `guard` 作为参数传入，第二个参数是一个函数，当这个函数执行的时候再递归执行 `step` 函数，前进到下一个，注意这里的 `fn` 就是我们刚才的 `iterator` 函数，那么我们再回到 `iterator` 函数的定义：

```

const iterator = (hook: NavigationGuard, next) => {
 if (this.pending !== route) {
 return abort()
 }
 try {
 hook(route, current, (to: any) => {
 if (to === false || isError(to)) {
 this.ensureURL(true)
 abort(to)
 } else if (
 typeof to === 'string' ||
 (typeof to === 'object' && (
 typeof to.path === 'string' ||
 typeof to.name === 'string'
)))
) {
 abort()
 if (typeof to === 'object' && to.replace) {
 this.replace(to)
 } else {
 this.push(to)
 }
 } else {
 next(to)
 }
 })
 } catch (e) {
 abort(e)
 }
}

```

`iterator` 函数逻辑很简单，它就是去执行每一个导航守卫 `hook`，并传入 `route`、`current` 和匿名函数，这些参数对应文档中的 `to`、`from`、`next`，当执行了匿名函数，会根据一些条件执行 `abort` 或 `next`，只有执行 `next` 的时候，才会前进到下一个导航守卫钩子函数中，这也就是为什么官方文档会说只有执行 `next` 方法来 `resolve` 这个钩子函数。

那么最后我们来看 `queue` 是怎么构造的：

```

const queue: Array<?NavigationGuard> = [] .concat(
 extractLeaveGuards(deactivated),
 this.router.beforeHooks,
 extractUpdateHooks(updated),
 activated.map(m => m.beforeEnter),
 resolveAsyncComponents(activated)
)

```

按照顺序如下：

1. 在失活的组件里调用离开守卫。
2. 调用全局的 `beforeEach` 守卫。
3. 在重用的组件里调用 `beforeRouteUpdate` 守卫
4. 在激活的路由配置里调用 `beforeEnter`。
5. 解析异步路由组件。

接下来我们来分别介绍这 5 步的实现。

第一步是通过执行 `extractLeaveGuards(deactivated)`，先来看一下 `extractLeaveGuards` 的定义：

```

function extractLeaveGuards (deactivated: Array<RouteRecord>): Array<?Function> {
 return extractGuards(deactivated, 'beforeRouteLeave', bindGuard, true)
}

```

它内部调用了 `extractGuards` 的通用方法，可以从 `RouteRecord` 数组中提取各个阶段的守卫：

```

function extractGuards (
 records: Array<RouteRecord>,
 name: string,
 bind: Function,
 reverse?: boolean
): Array<?Function> {
 const guards = flatMapComponents(records, (def, instance, match, key) => {
 const guard = extractGuard(def, name)
 if (guard) {
 return Array.isArray(guard)
 ? guard.map(guard => bind(guard, instance, match, key))
 : bind(guard, instance, match, key)
 }
 })
 return flatten(reverse ? guards.reverse() : guards)
}

```

这里用到了 `flatMapComponents` 方法去从 `records` 中获取所有的导航，它的定义在 `src/util/resolve-components.js` 中：

```
export function flatMapComponents (
 matched: Array<RouteRecord>,
 fn: Function
): Array<?Function> {
 return flatten(matched.map(m => {
 return Object.keys(m.components).map(key => fn(
 m.components[key],
 m.instances[key],
 m, key
))
 }))
}

export function flatten (arr: Array<any>): Array<any> {
 return Array.prototype.concat.apply([], arr)
}
```

`flatMapComponents` 的作用就是返回一个数组，数组的元素是从 `matched` 里获取到所有组件的 `key`，然后返回 `fn` 函数执行的结果，`flatten` 作用是把二维数组拍平成一维数组。

那么对于 `extractGuards` 中 `flatMapComponents` 的调用，执行每个 `fn` 的时候，通过 `extractGuard(def, name)` 获取到组件中对应 `name` 的导航守卫：

```
function extractGuard (
 def: Object | Function,
 key: string
): NavigationGuard | Array<NavigationGuard> {
 if (typeof def !== 'function') {
 def = _Vue.extend(def)
 }
 return def.options[key]
}
```

获取到 `guard` 后，还会调用 `bind` 方法把组件的实例 `instance` 作为函数执行的上下文绑定到 `guard` 上，`bind` 方法的对应的是 `bindGuard`：

```
function bindGuard (guard: NavigationGuard, instance: ?_Vue): ?NavigationGuard {
 if (instance) {
 return function boundRouteGuard () {
 return guard.apply(instance, arguments)
 }
 }
}
```

那么对于 `extractLeaveGuards(deactivated)` 而言，获取到的就是所有失活组件中定义的 `beforeRouteLeave` 钩子函数。

第二步是 `this.router.beforeHooks`，在我们的 `VueRouter` 类中定义了 `beforeEach` 方法，在 `src/index.js` 中：

```
beforeEach (fn: Function): Function {
 return registerHook(this.beforeHooks, fn)
}

function registerHook (list: Array<any>, fn: Function): Function {
 list.push(fn)
 return () => {
 const i = list.indexOf(fn)
 if (i > -1) list.splice(i, 1)
 }
}
```

当用户使用 `router.beforeEach` 注册了一个全局守卫，就会往 `router.beforeHooks` 添加一个钩子函数，这样 `this.router.beforeHooks` 获取的就是用户注册的全局 `beforeEach` 守卫。

第三步执行了 `extractUpdateHooks(updated)`，来看一下 `extractUpdateHooks` 的定义：

```
function extractUpdateHooks (updated: Array<RouteRecord>): Array<?Function> {
 return extractGuards(updated, 'beforeRouteUpdate', bindGuard)
}
```

和 `extractLeaveGuards(deactivated)` 类似，`extractUpdateHooks(updated)` 获取到的就是所有重用的组件中定义的 `beforeRouteUpdate` 钩子函数。

第四步是执行 `activated.map(m => m.beforeEnter)`，获取的是在激活的路由配置中定义的 `beforeEnter` 函数。

第五步是执行 `resolveAsyncComponents(activated)` 解析异步组件，先来看一下 `resolveAsyncComponents` 的定义，在 `src/util/resolve-components.js` 中：

```
export function resolveAsyncComponents (matched: Array<RouteRecord>): Function {
 return (to, from, next) => {
 let hasAsync = false
 let pending = 0
 let error = null

 flatMapComponents(matched, (def, _, match, key) => {
 if (typeof def === 'function' && def.cid === undefined) {
 hasAsync = true
 pending++
 }

 const resolve = once(resolvedDef => {
 if (isESModule(resolvedDef)) {
```

```

 resolvedDef = resolvedDef.default
 }
 def.resolved = typeof resolvedDef === 'function'
 ? resolvedDef
 : _Vue.extend(resolvedDef)
match.components[key] = resolvedDef
pending--
if (pending <= 0) {
 next()
}
})

const reject = once(reason => {
 const msg = `Failed to resolve async component ${key}: ${reason}`
 process.env.NODE_ENV !== 'production' && warn(false, msg)
 if (!error) {
 error = isError(reason)
 ? reason
 : new Error(msg)
 next(error)
 }
})

let res
try {
 res = def(resolve, reject)
} catch (e) {
 reject(e)
}
if (res) {
 if (typeof res.then === 'function') {
 res.then(resolve, reject)
 } else {
 const comp = res.component
 if (comp && typeof comp.then === 'function') {
 comp.then(resolve, reject)
 }
 }
}
}

if (!hasAsync) next()
}
}

```

`resolveAsyncComponents` 返回的是一个导航守卫函数，有标准的 `to`、`from`、`next` 参数。它的内部实现很简单，利用了 `flatMapComponents` 方法从 `matched` 中获取到每个组件的定义，判断如果是异步组件，则执行异步组件加载逻辑，这块和我们之前分析 `vue` 加载异步组件很类似，加载

成功后会执行 `match.components[key] = resolvedDef` 把解析好的异步组件放到对应的 `components` 上，并且执行 `next` 函数。

这样在 `resolveAsyncComponents(activated)` 解析完所有激活的异步组件后，我们就可以拿到这一次所有激活的组件。这样我们在做完这 5 步后又做了一些事情：

```
runQueue(queue, iterator, () => {
 const postEnterCbs = []
 const isValid = () => this.current === route
 const enterGuards = extractEnterGuards(activated, postEnterCbs, isValid)
 const queue = enterGuards.concat(this.router.resolveHooks)
 runQueue(queue, iterator, () => {
 if (this.pending !== route) {
 return abort()
 }
 this.pending = null
 onComplete(route)
 if (this.router.app) {
 this.router.app.$nextTick(() => {
 postEnterCbs.forEach(cb => { cb() })
 })
 }
 })
})
```

1. 在被激活的组件里调用 `beforeRouteEnter`。
2. 调用全局的 `beforeResolve` 守卫。
3. 调用全局的 `afterEach` 钩子。

对于第六步有这些相关的逻辑：

```
const postEnterCbs = []
const isValid = () => this.current === route
const enterGuards = extractEnterGuards(activated, postEnterCbs, isValid)

function extractEnterGuards (
 activated: Array<RouteRecord>,
 cbs: Array<Function>,
 isValid: () => boolean
): Array<?Function> {
 return extractGuards(activated, 'beforeRouteEnter', (guard, _, match, key) => {
 return bindEnterGuard(guard, match, key, cbs, isValid)
 })
}

function bindEnterGuard (
 guard: NavigationGuard,
 match: RouteRecord,
```

```

key: string,
cbs: Array<Function>,
isValid: () => boolean
): NavigationGuard {
 return function routeEnterGuard (to, from, next) {
 return guard(to, from, cb => {
 next(cb)
 if (typeof cb === 'function') {
 cbs.push(() => {
 poll(cb, match.instances, key, isValid)
 })
 }
 })
 }
}

function poll (
 cb: any,
 instances: Object,
 key: string,
 isValid: () => boolean
) {
 if (instances[key]) {
 cb(instances[key])
 } else if (isValid()) {
 setTimeout(() => {
 poll(cb, instances, key, isValid)
 }, 16)
 }
}

```

`extractEnterGuards` 函数的实现也是利用了 `extractGuards` 方法提取组件中的 `beforeRouteEnter` 导航钩子函数，和之前不同的是 `bind` 方法的不同。文档中特意强调了 `beforeRouteEnter` 钩子函数中是拿不到组件实例的，因为当守卫执行前，组件实例还没被创建，但是我们可以通过传一个回调给 `next` 来访问组件实例。在导航被确认的时候执行回调，并且把组件实例作为回调方法的参数：

```

beforeRouteEnter (to, from, next) {
 next(vm => {
 // 通过 `vm` 访问组件实例
 })
}

```

来看一下这是怎么实现的。

在 `bindEnterGuard` 函数中，返回的是 `routeEnterGuard` 函数，所以在执行 `iterator` 中的 `hook` 函数的时候，就相当于执行 `routeEnterGuard` 函数，那么就会执行我们定义的导航守卫 `guard` 函数，并且当这个回调函数执行的时候，首先执行 `next` 函数 `resolver` 当前导航钩子，

然后把回调函数的参数，它也是一个回调函数用 `cbs` 收集起来，其实就是收集到外面定义的 `postEnterCbs` 中，然后在最后会执行：

```
if (this.router.app) {
 this.router.app.$nextTick(() => {
 postEnterCbs.forEach(cb => { cb() })
 })
}
```

在根路由组件重新渲染后，遍历 `postEnterCbs` 执行回调，每一个回调执行的时候，其实是执行 `poll(cb, match.instances, key, isValid)` 方法，因为考虑到一些路由组件被套 `transition` 组件在一些缓动模式下不一定能拿到实例，所以用一个轮询方法不断去判断，直到能获取到组件实例，再去调用 `cb`，并把组件实例作为参数传入，这就是我们在回调函数中能拿到组件实例的原因。

第七步是获取 `this.router.resolveHooks`，这个和 `this.router.beforeHooks` 的获取类似，在我们的 `VueRouter` 类中定义了 `beforeResolve` 方法：

```
beforeResolve (fn: Function): Function {
 return registerHook(this.resolveHooks, fn)
}
```

当用户使用 `router.beforeResolve` 注册了一个全局守卫，就会往 `router.resolveHooks` 添加一个钩子函数，这样 `this.router.resolveHooks` 获取的就是用户注册的全局 `beforeResolve` 守卫。

第八步是在最后执行了 `onComplete(route)` 后，会执行 `this.updateRoute(route)` 方法：

```
updateRoute (route: Route) {
 const prev = this.current
 this.current = route
 this.cb && this.cb(route)
 this.router.afterHooks.forEach(hook => {
 hook && hook(route, prev)
 })
}
```

同样在我们的 `VueRouter` 类中定义了 `afterEach` 方法：

```
afterEach (fn: Function): Function {
 return registerHook(this.afterHooks, fn)
}
```

当用户使用 `router.afterEach` 注册了一个全局守卫，就会往 `router.afterHooks` 添加一个钩子函数，这样 `this.router.afterHooks` 获取的就是用户注册的全局 `afterHooks` 守卫。

那么至此我们把所有导航守卫的执行分析完毕了，我们知道路由切换除了执行这些钩子函数，从表象上有 2 个地方会发生变化，一个是 url 发生变化，一个是组件发生变化。接下来我们分别介绍这两块的实现原理。

## url

在 `confirmTransition` 的 `onComplete` 函数中，在 `updateRoute` 后，会执行 `this.ensureURL()` 函数，这个函数是子类实现的，不同模式下该函数的实现略有不同，我们来看一下平时使用最多的 `hash` 模式该函数的实现，在 `src/history/hash.js` 中。

```
ensureURL (push?: boolean) {
 const current = this.current fullPath
 if (getHash() !== current) {
 push ? pushHash(current) : replaceHash(current)
 }
}

export function getHash (): string {
 const href = window.location.href
 const index = href.indexOf('#')
 return index === -1 ? '' : href.slice(index + 1)
}

function getUrl (path) {
 const href = window.location.href
 const i = href.indexOf('#')
 const base = i >= 0 ? href.slice(0, i) : href
 return `${base}#${path}`
}

function pushHash (path) {
 if (supportsPushState) {
 pushState(getUrl(path))
 } else {
 window.location.hash = path
 }
}

function replaceHash (path) {
 if (supportsPushState) {
 replaceState(getUrl(path))
 } else {
 window.location.replace(getUrl(path))
 }
}
```

`ensureURL` 函数首先判断当前 `hash` 和当前的券路径是否相等，如果不相等，则根据 `push` 参数决定执行 `pushHash` 或者是 `replaceHash`。

`supportsPushState` 的定义在 `src/util/push-state.js` 中：

```
export const supportsPushState = inBrowser && (function () {
 const ua = window.navigator.userAgent

 if (
 (ua.indexOf('Android 2.') !== -1 || ua.indexOf('Android 4.0') !== -1) &&
 ua.indexOf('Mobile Safari') !== -1 &&
 ua.indexOf('Chrome') === -1 &&
 ua.indexOf('Windows Phone') === -1
) {
 return false
 }

 return window.history && 'pushState' in window.history
})()
```

如果支持的话，则获取当前完整的 `url`，执行 `pushState` 方法：

```
export function pushState (url?: string, replace?: boolean) {
 saveScrollPosition()
 const history = window.history
 try {
 if (replace) {
 history.replaceState({ key: '_key' }, '', url)
 } else {
 _key = genKey()
 history.pushState({ key: '_key' }, '', url)
 }
 } catch (e) {
 window.location[replace ? 'replace' : 'assign'](url)
 }
}
```

`pushState` 会调用浏览器原生的 `history` 的 `pushState` 接口或者 `replaceState` 接口，更新浏览器的 `url` 地址，并把当前 `url` 压入历史栈中。

然后在 `history` 的初始化中，会设置一个监听器，监听历史栈的变化：

```
setupListeners () {
 const router = this.router
 const expectScroll = router.options.scrollBehavior
 const supportsScroll = supportsPushState && expectScroll

 if (supportsScroll) {
 setupScroll()
 }

 window.addEventListener(supportsPushState ? 'popstate' : 'hashchange', () => {
```

```

const current = this.current
if (!ensureSlash()) {
 return
}
this.transitionTo(getHash(), route => {
 if (supportsScroll) {
 handleScroll(this.router, route, current, true)
 }
 if (!supportsPushState) {
 replaceHash(route fullPath)
 }
})
})
}
}

```

当点击浏览器返回按钮的时候，如果有 url 被压入历史栈，则会触发 `popstate` 事件，然后拿到当前要跳转的 `hash`，执行 `transitionTo` 方法做一次路径转换。

同学们在使用 Vue-Router 开发项目的时候，打开调试页面 `http://localhost:8080` 后会自动把 url 修改为 `http://localhost:8080/#/`，这是怎么做到呢？原来在实例化 `HashHistory` 的时候，构造函数会执行 `ensureSlash()` 方法：

```

function ensureSlash (): boolean {
 const path = getHash()
 if (path.charAt(0) === '/') {
 return true
 }
 replaceHash('/' + path)
 return false
}

```

这个时候 `path` 为空，所以执行 `replaceHash('/' + path)`，然后内部会执行一次 `getUrl`，计算出来的新的 url 为 `http://localhost:8080/#/`，这就是 url 会改变的原因。

## 组件

路由最终的渲染离不开组件，Vue-Router 内置了 `<router-view>` 组件，它的定义在 `src/components/view.js` 中。

```

export default {
 name: 'RouterView',
 functional: true,
 props: {
 name: {
 type: String,
 default: 'default'
 }
 },
}

```

```

render (_, { props, children, parent, data }) {
 data.routerView = true

 const h = parent.$createElement
 const name = props.name
 const route = parent.$route
 const cache = parent._routerViewCache || (parent._routerViewCache = {})

 let depth = 0
 let inactive = false
 while (parent && parent._routerRoot !== parent) {
 if (parent.$vnode && parent.$vnode.data.routerView) {
 depth++
 }
 if (parent._inactive) {
 inactive = true
 }
 parent = parent.$parent
 }
 data.routerViewDepth = depth

 if (inactive) {
 return h(cache[name], data, children)
 }

 const matched = route.matched[depth]
 if (!matched) {
 cache[name] = null
 return h()
 }

 const component = cache[name] = matched.components[name]

 data.registerRouteInstance = (vm, val) => {
 const current = matched.instances[name]
 if (
 (val && current !== vm) ||
 (!val && current === vm)
) {
 matched.instances[name] = val
 }
 }

 ;(data.hook || (data.hook = {})).prepatch = (_, vnode) => {
 matched.instances[name] = vnode.componentInstance
 }

 let propsToPass = data.props = resolveProps(route, matched.props && matched.props[name])
 if (propsToPass) {
 propsToPass = data.props = extend({}, propsToPass)
 }
}

```

```

 const attrs = data.attrs = data.attrs || {}
 for (const key in propsToPass) {
 if (!component.props || !(key in component.props)) {
 attrs[key] = propsToPass[key]
 delete propsToPass[key]
 }
 }
 }

 return h(component, data, children)
}
}

```

`<router-view>` 是一个 `functional` 组件，它的渲染也是依赖 `render` 函数，那么 `<router-view>` 具体应该渲染什么组件呢，首先获取当前的路径：

```
const route = parent.$route
```

我们之前分析过，在 `src/install.js` 中，我们给 `Vue` 的原型上定义了 `$route`：

```

Object.defineProperty(Vue.prototype, '$route', {
 get () { return this._routerRoot._route }
})

```

然后在 `VueRouter` 的实例执行 `router.init` 方法的时候，会执行如下逻辑，定义在 `src/index.js` 中：

```

history.listen(route => {
 this.apps.forEach((app) => {
 app._route = route
 })
})

```

而 `history.listen` 方法定义在 `src/history/base.js` 中：

```

listen (cb: Function) {
 this.cb = cb
}

```

然后在 `updateRoute` 的时候执行 `this.cb`：

```

updateRoute (route: Route) {
 // ...
 this.current = route
 this.cb && this.cb(route)
 // ...
}

```

```
}
```

也就是我们执行 `transitionTo` 方法最后执行 `updateRoute` 的时候会执行回调，然后会更新所有组件实例的 `_route` 值，所以说 `$route` 对应的就是当前的路由线路。

`<router-view>` 是支持嵌套的，回到 `render` 函数，其中定义了 `depth` 的概念，它表示 `<router-view>` 嵌套的深度。每个 `<router-view>` 在渲染的时候，执行如下逻辑：

```
data.routerView = true
// ...
while (parent && parent._routerRoot !== parent) {
 if (parent.$vnode && parent.$vnode.data.routerView) {
 depth++
 }
 if (parent._inactive) {
 inactive = true
 }
 parent = parent.$parent
}

const matched = route.matched[depth]
// ...
const component = cache[name] = matched.components[name]
```

`parent._routerRoot` 表示的是根 Vue 实例，那么这个循环就是从当前的 `<router-view>` 的父节点向上找，一直找到根 Vue 实例，在这个过程，如果碰到了父节点也是 `<router-view>` 的时候，说明 `<router-view>` 有嵌套的情况，`depth++`。遍历完成后，根据当前线路匹配的路径和 `depth` 找到对应的 `RouteRecord`，进而找到该渲染的组件。

除了找到了应该渲染的组件，还定义了一个注册路由实例的方法：

```
data.registerRouteInstance = (vm, val) => {
 const current = matched.instances[name]
 if (
 (val && current !== vm) ||
 (!val && current === vm)
) {
 matched.instances[name] = val
 }
}
```

给 `vnode` 的 `data` 定义了 `registerRouteInstance` 方法，在 `src/install.js` 中，我们会调用该方法去注册路由的实例：

```
const registerInstance = (vm, callVal) => {
 let i = vm.$options._parentVnode
 if (isDef(i) && isDef(i = i.data) && isDef(i = i.registerRouteInstance)) {
 i(vm, callVal)
```

```

 }
 }

Vue.mixin({
 beforeCreate () {
 // ...
 registerInstance(this, this)
 },
 destroyed () {
 registerInstance(this)
 }
})

```

在混入的 `beforeCreate` 钩子函数中，会执行 `registerInstance` 方法，进而执行 `render` 函数中定义的 `registerRouteInstance` 方法，从而给 `matched.instances[name]` 赋值当前组件的 `vm` 实例。

`render` 函数的最后根据 `component` 渲染出对应的组件 `node`：

```
return h(component, data, children)
```

那么当我们执行 `transitionTo` 来更改路由线路后，组件是如何重新渲染的呢？在我们混入的 `beforeCreated` 钩子函数中有这么一段逻辑：

```

Vue.mixin({
 beforeCreate () {
 if (isDef(this.$options.router)) {
 Vue.util.defineReactive(this, '_route', this._router.history.current)
 }
 // ...
 }
})

```

由于我们把根 `Vue` 实例的 `_route` 属性定义成响应式的，我们在每个 `<router-view>` 执行 `render` 函数的时候，都会访问 `parent.$route`，如我们之前分析会访问 `this._routerRoot._route`，触发了它的 `getter`，相当于 `<router-view>` 对它有依赖，然后再执行完 `transitionTo` 后，修改 `app._route` 的时候，又触发了 `setter`，因此会通知 `<router-view>` 的渲染 `watcher` 更新，重新渲染组件。

`Vue-Router` 还内置了另一个组件 `<router-link>`，它支持用户在具有路由功能的应用中（点击）导航。通过 `to` 属性指定目标地址，默认渲染成带有正确链接的 `<a>` 标签，可以通过配置 `tag` 属性生成别的标签。另外，当目标路由成功激活时，链接元素自动设置一个表示激活的 CSS 类名。

`<router-link>` 比起写死的 `<a href="...">` 会好一些，理由如下：

无论是 `HTML5 history` 模式还是 `hash` 模式，它的表现行为一致，所以，当你要切换路由模式，或者在 IE9 降级使用 `hash` 模式，无须作任何变动。

在 HTML5 `history` 模式下，`router-link` 会守卫点击事件，让浏览器不再重新加载页面。

当你在 HTML5 `history` 模式下使用 `base` 选项之后，所有的 `to` 属性都不需要写（基路径）了。

那么接下来我们就来分析它的实现，它的定义在 `src/components/link.js` 中：

```

export default {
 name: 'RouterLink',
 props: {
 to: {
 type: toTypes,
 required: true
 },
 tag: {
 type: String,
 default: 'a'
 },
 exact: Boolean,
 append: Boolean,
 replace: Boolean,
 activeClass: String,
 exactActiveClass: String,
 event: {
 type: eventTypes,
 default: 'click'
 }
 },
 render (h: Function) {
 const router = this.$router
 const current = this.$route
 const { location, route, href } = router.resolve(this.to, current, this.append)

 const classes = {}
 const globalActiveClass = router.options.linkActiveClass
 const globalExactActiveClass = router.options.linkExactActiveClass
 const activeClassFallback = globalActiveClass == null
 ? 'router-link-active'
 : globalActiveClass
 const exactActiveClassFallback = globalExactActiveClass == null
 ? 'router-link-exact-active'
 : globalExactActiveClass
 const activeClass = this.activeClass == null
 ? activeClassFallback
 : this.activeClass
 const exactActiveClass = this.exactActiveClass == null
 ? exactActiveClassFallback
 : this.exactActiveClass
 const compareTarget = location.path
 ? createRoute(null, location, null, router)
 : route
 }
}

```

```

classes[exactActiveClass] = isSameRoute(current, compareTarget)
classes[activeClass] = this.exact
? classes[exactActiveClass]
: isIncludedRoute(current, compareTarget)

const handler = e => {
 if (guardEvent(e)) {
 if (this.replace) {
 router.replace(location)
 } else {
 router.push(location)
 }
 }
}

const on = { click: guardEvent }
if (Array.isArray(this.event)) {
 this.event.forEach(e => { on[e] = handler })
} else {
 on[this.event] = handler
}

const data: any = {
 class: classes
}

if (this.tag === 'a') {
 data.on = on
 data.attrs = { href }
} else {
 const a = findAnchor(this.$slots.default)
 if (a) {
 a.isStatic = false
 const extend = _Vue.util.extend
 const aData = a.data = extend({}, a.data)
 aData.on = on
 const aAttrs = a.data.attrs = extend({}, a.data.attrs)
 aAttrs.href = href
 } else {
 data.on = on
 }
}

return h(this.tag, data, this.$slots.default)
}
}

```

<router-link> 标签的渲染也是基于 `render` 函数，它首先做了路由解析：

```
const router = this.$router
```

```
const current = this.$route
const { location, route, href } = router.resolve(this.to, current, this.append)
```

`router.resolve` 是 `VueRouter` 的实例方法，它的定义在 `src/index.js` 中：

```
resolve (
 to: RawLocation,
 current?: Route,
 append?: boolean
): {
 location: Location,
 route: Route,
 href: string,
 normalizedTo: Location,
 resolved: Route
} {
 const location = normalizeLocation(
 to,
 current || this.history.current,
 append,
 this
)
 const route = this.match(location, current)
 const fullPath = route.redirectedFrom || route.fullPath
 const base = this.history.base
 const href = createHref(base, fullPath, this.mode)
 return {
 location,
 route,
 href,
 normalizedTo: location,
 resolved: route
 }
}

function createHref (base: string, fullPath: string, mode) {
 var path = mode === 'hash' ? '#' + fullPath : fullPath
 return base ? cleanPath(base + '/' + path) : path
}
```

它先规范生成目标 `location`，再根据 `location` 和 `match` 通过 `this.match` 方法计算生成目标路径 `route`，然后再根据 `base`、`fullPath` 和 `this.mode` 通过 `createHref` 方法计算出最终跳转的 `href`。

解析完 `router` 获得目标 `location`、`route`、`href` 后，接下来对 `exactActiveClass` 和 `activeClass` 做处理，当配置 `exact` 为 `true` 的时候，只有当目标路径和当前路径完全匹配的时候，会添加 `exactActiveClass`；而当目标路径包含当前路径的时候，会添加 `activeClass`。

接着创建了一个守卫函数：

```

const handler = e => {
 if (guardEvent(e)) {
 if (this.replace) {
 router.replace(location)
 } else {
 router.push(location)
 }
 }
}

function guardEvent (e) {
 if (e.metaKey || e.altKey || e.ctrlKey || e.shiftKey) return
 if (e.defaultPrevented) return
 if (e.button !== undefined && e.button !== 0) return
 if (e.currentTarget && e.currentTarget.getAttribute) {
 const target = e.currentTarget.getAttribute('target')
 if (/^_blank$/i.test(target)) return
 }
 if (e.preventDefault) {
 e.preventDefault()
 }
 return true
}

const on = { click: guardEvent }
if (Array.isArray(this.event)) {
 this.event.forEach(e => { on[e] = handler })
} else {
 on[this.event] = handler
}

```

最终会监听点击事件或者其它可以通过 `prop` 传入的事件类型，执行 `handler` 函数，最终执行 `router.push` 或者 `router.replace` 函数，它们的定义在 `src/index.js` 中：

```

push (location: RawLocation, onComplete?: Function, onAbort?: Function) {
 this.history.push(location, onComplete, onAbort)
}

replace (location: RawLocation, onComplete?: Function, onAbort?: Function) {
 this.history.replace(location, onComplete, onAbort)
}

```

实际上就是执行了 `history` 的 `push` 和 `replace` 方法做路由跳转。

最后判断当前 `tag` 是否是 `<a>` 标签，`<router-link>` 默认会渲染成 `<a>` 标签，当然我们也可以修改 `tag` 的 `prop` 渲染成其他节点，这种情况下会尝试找它子元素的 `<a>` 标签，如果有则把事件绑定到 `<a>` 标签上并添加 `href` 属性，否则绑定到外层元素本身。

## 总结

那么至此我们把路由的 `transitionTo` 的主体过程分析完毕了，其他一些分支比如重定向、别名、滚动行为等同学们可以自行再去分析。

路径变化是路由中最重要的功能，我们要记住以下内容：路由始终会维护当前的线路，路由切换的时候会把当前线路切换到目标线路，切换过程中会执行一系列的导航守卫钩子函数，会更改 url，同样也会渲染对应的组件，切换完毕后会把目标线路更新替换当前线路，这样就会作为下一次的路径切换的依据。

# Vuex

Vuex 是一个专为 Vue.js 应用程序开发的状态管理模式。它采用集中式存储管理应用的所有组件的状态，并以相应的规则保证状态以一种可预测的方式发生变化。

## 什么是“状态管理模式”？

让我们从一个简单的 Vue 计数应用开始：

```
new Vue({
 // state
 data () {
 return {
 count: 0
 }
 },
 // view
 template: `
 <div>{{ count }}</div>
 `,
 // actions
 methods: {
 increment () {
 this.count++
 }
 }
})
```

这个状态自管理应用包含以下几个部分：

- state， 驱动应用的数据源；
- view， 以声明方式将 state 映射到视图；
- actions， 响应在 view 上的用户输入导致的状态变化。

以下是一个表示“单向数据流”理念的极简示意：

```
// TODO 图片
```

但是，当我们的应用遇到多个组件共享状态时，单向数据流的简洁性很容易被破坏：

- 多个视图依赖于同一状态。
- 来自不同视图的行为需要变更同一状态。

对于问题一，传参的方法对于多层嵌套的组件将会非常繁琐，并且对于兄弟组件间的状态传递无能为力。对于问题二，我们经常会采用父子组件直接引用或者通过事件来变更和同步状态的多份拷贝。以上的这些模式非常脆弱，通常会导致无法维护的代码。

因此，我们为什么不把组件的共享状态抽取出来，以一个全局单例模式管理呢？在这种模式下，我们的组件树构成了一个巨大的“视图”，不管在树的哪个位置，任何组件都能获取状态或者触发行为。

## Vuex 核心思想

Vuex 应用的核心就是 store（仓库）。 “store”基本上就是一个容器，它包含着你的应用中大部分的状态（state）。有些同学可能会问，那我定义一个全局对象，再去上层封装了一些数据存取的接口不可以么？

Vuex 和单纯的全局对象有以下两点不同：

- Vuex 的状态存储是响应式的。当 Vue 组件从 store 中读取状态的时候，若 store 中的状态发生变化，那么相应的组件也会相应地得到高效更新。
- 你不能直接改变 store 中的状态。改变 store 中的状态的唯一途径就是显式地提交（commit）mutation。这样使得我们可以方便地跟踪每一个状态的变化，从而让我们能够实现一些工具帮助我们更好地了解我们的应用。

另外，通过定义和隔离状态管理中的各种概念并强制遵守一定的规则，我们的代码将会变得更结构化且易维护。

// TODO 图片

# Vuex 初始化

这一节我们主要来分析 Vuex 的初始化过程，它包括安装、Store 实例化过程 2 个方面。

## 安装

当我们在代码中通过 `import Vuex from 'vuex'` 的时候，实际上引用的是一个对象，它的定义在 `src/index.js` 中：

```
export default {
 Store,
 install,
 version: '__VERSION__',
 mapState,
 mapMutations,
 mapGetters,
 mapActions,
 createNamespacedHelpers
}
```

和 Vue-Router 一样，Vuex 也同样存在一个静态的 `install` 方法，它的定义在 `src/store.js` 中：

```
export function install (_Vue) {
 if (Vue && _Vue === Vue) {
 if (process.env.NODE_ENV !== 'production') {
 console.error(
 '[vuex] already installed. Vue.use(Vuex) should be called only once.'
)
 }
 return
 }
 Vue = _Vue
 applyMixin(Vue)
}
```

`install` 的逻辑很简单，把传入的 `_Vue` 赋值给 `Vue` 并执行了 `applyMixin(Vue)` 方法，它的定义在 `src/mixin.js` 中：

```
export default function (Vue) {
 const version = Number(Vue.version.split('.')[0])

 if (version >= 2) {
 Vue.mixin({ beforeCreate: vuexInit })
 } else {
 // override init and inject vuex init procedure
 // for 1.x backwards compatibility.
 }
}
```

```

const _init = Vue.prototype._init
Vue.prototype._init = function (options = {}) {
 options.init = options.init
 ? [vuexInit].concat(options.init)
 : vuexInit
 _init.call(this, options)
}

/**
 * Vuex init hook, injected into each instances init hooks list.
 */

function vuexInit () {
 const options = this.$options
 // store injection
 if (options.store) {
 this.$store = typeof options.store === 'function'
 ? options.store()
 : options.store
 } else if (options.parent && options.parent.$store) {
 this.$store = options.parent.$store
 }
}

```

`applyMixin` 就是这个 `export default function`，它还兼容了 Vue 1.0 的版本，这里我们只关注 Vue 2.0 以上版本的逻辑，它其实就全局混入了一个 `beforeCreated` 钩子函数，它的实现非常简单，就是把 `options.store` 保存在所有组件的 `this.$store` 中，这个 `options.store` 就是我们在实例化 `Store` 对象的实例，稍后我们会介绍，这也是为什么我们在组件中可以通过 `this.$store` 访问到这个实例。

## Store 实例化

我们在 `import Vuex` 之后，会实例化其中的 `Store` 对象，返回 `store` 实例并传入 `new Vue` 的 `options` 中，也就是我们刚才提到的 `options.store`。

举个简单的例子，如下：

```

export default new Vuex.Store({
 actions,
 getters,
 state,
 mutations,
 modules
 // ...
})

```

`Store` 对象的构造函数接收一个对象参数，它包含

`actions`、`getters`、`state`、`mutations`、`modules` 等 Vuex 的核心概念，它的定义在 `src/store.js` 中：

```
export class Store {
 constructor (options = {}) {
 // Auto install if it is not done yet and `window` has `Vue`.
 // To allow users to avoid auto-installation in some cases,
 // this code should be placed here. See #731
 if (!Vue && typeof window !== 'undefined' && window.Vue) {
 install(window.Vue)
 }

 if (process.env.NODE_ENV !== 'production') {
 assert(Vue, `must call Vue.use(Vuex) before creating a store instance.`)
 assert(typeof Promise !== 'undefined', `vuex requires a Promise polyfill in this browser.`)
 assert(this instanceof Store, `Store must be called with the new operator.`)
 }

 const {
 plugins = [],
 strict = false
 } = options

 // store internal state
 this._committing = false
 this._actions = Object.create(null)
 this._actionSubscribers = []
 this._mutations = Object.create(null)
 this._wrappedGetters = Object.create(null)
 this._modules = new ModuleCollection(options)
 this._modulesNamespaceMap = Object.create(null)
 this._subscribers = []
 this._watcherVM = new Vue()

 // bind commit and dispatch to self
 const store = this
 const { dispatch, commit } = this
 this.dispatch = function boundDispatch (type, payload) {
 return dispatch.call(store, type, payload)
 }
 this.commit = function boundCommit (type, payload, options) {
 return commit.call(store, type, payload, options)
 }

 // strict mode
 this.strict = strict

 const state = this._modules.root.state
 }
}
```

```

// init root module.
// this also recursively registers all sub-modules
// and collects all module getters inside this._wrappedGetters
installModule(this, state, [], this._modules.root)

// initialize the store vm, which is responsible for the reactivity
// (also registers _wrappedGetters as computed properties)
resetStoreVM(this, state)

// apply plugins
plugins.forEach(plugin => plugin(this))

if (Vue.config.devtools) {
 devtoolPlugin(this)
}
}
}

```

我们把 `store` 的实例化过程拆成 3 个部分，分别是初始化模块，安装模块和初始化 `store._vm`，接下来我们来分析这 3 部分的实现。

## 初始化模块

在分析模块初始化之前，我们先来了解一下模块对于 Vuex 的意义：由于使用单一状态树，应用的所有状态会集中到一个比较大的对象，当应用变得非常复杂时，`store` 对象就有可能变得相当臃肿。为了解决以上问题，Vuex 允许我们将 `store` 分割成模块（module）。每个模块拥有自己的 `state`、`mutation`、`action`、`getter`，甚至是嵌套子模块——从上至下进行同样方式的分割：

```

const moduleA = {
 state: { ... },
 mutations: { ... },
 actions: { ... },
 getters: { ... }
}

const moduleB = {
 state: { ... },
 mutations: { ... },
 actions: { ... },
 getters: { ... },
}

const store = new Vuex.Store({
 modules: {
 a: moduleA,
 b: moduleB
 }
}

```

```

 })

store.state.a // -> moduleA 的状态
store.state.b // -> moduleB 的状态

```

所以从数据结构上来看，模块的设计就是一个树型结构，`store` 本身可以理解为一个 `root module`，它下面的 `modules` 就是子模块，Vuex 需要完成这颗树的构建，构建过程的入口就是：

```
this._modules = new ModuleCollection(options)
```

`ModuleCollection` 的定义在 `src/module/module-collection.js` 中：

```

export default class ModuleCollection {
 constructor (rawRootModule) {
 // register root module (Vuex.Store options)
 this.register([], rawRootModule, false)
 }

 get (path) {
 return path.reduce((module, key) => {
 return module.getChild(key)
 }, this.root)
 }

 getNamespace (path) {
 let module = this.root
 return path.reduce((namespace, key) => {
 module = module.getChild(key)
 return namespace + (module.namespaced ? key + '/' : '') + ''
 }, '')
 }

 update (rawRootModule) {
 update([], this.root, rawRootModule)
 }

 register (path, rawModule, runtime = true) {
 if (process.env.NODE_ENV !== 'production') {
 assertRawModule(path, rawModule)
 }

 const newModule = new Module(rawModule, runtime)
 if (path.length === 0) {
 this.root = newModule
 } else {
 const parent = this.get(path.slice(0, -1))
 parent.addChild(path[path.length - 1], newModule)
 }
 }
}

```

```

// register nested modules
if (rawModule.modules) {
 forEachValue(rawModule.modules, (rawChildModule, key) => {
 this.register(path.concat(key), rawChildModule, runtime)
 })
}
}

unregister (path) {
 const parent = this.get(path.slice(0, -1))
 const key = path[path.length - 1]
 if (!parent.getChild(key).runtime) return

 parent.removeChild(key)
}
}

```

`ModuleCollection` 实例化的过程就是执行了 `register` 方法，`register` 接收 3 个参数，其中 `path` 表示路径，因为我们整体目标是要构建一颗模块树，`path` 是在构建树的过程中维护的路径；`rawModule` 表示定义模块的原始配置；`runtime` 表示是否是一个运行时创建的模块。

`register` 方法首先通过 `const newModule = new Module(rawModule, runtime)` 创建了一个 `Module` 的实例，`Module` 是用来描述单个模块的类，它的定义在 `src/module/module.js` 中：

```

export default class Module {
 constructor (rawModule, runtime) {
 this.runtime = runtime
 // Store some children item
 this._children = Object.create(null)
 // Store the origin module object which passed by programmer
 this._rawModule = rawModule
 const rawState = rawModule.state

 // Store the origin module's state
 this.state = (typeof rawState === 'function' ? rawState() : rawState) || {}
 }

 get namespaced () {
 return !!this._rawModule.namespaced
 }

 addChild (key, module) {
 this._children[key] = module
 }

 removeChild (key) {
 delete this._children[key]
 }

 getChild (key) {

```

```

 return this._children[key]
 }

 update (rawModule) {
 this._rawModule.namespaced = rawModule.namespaced
 if (rawModule.actions) {
 this._rawModule.actions = rawModule.actions
 }
 if (rawModule.mutations) {
 this._rawModule.mutations = rawModule.mutations
 }
 if (rawModule.getters) {
 this._rawModule.getters = rawModule.getters
 }
 }

 forEachChild (fn) {
 forEachValue(this._children, fn)
 }

 forEachGetter (fn) {
 if (this._rawModule.getters) {
 forEachValue(this._rawModule.getters, fn)
 }
 }

 forEachAction (fn) {
 if (this._rawModule.actions) {
 forEachValue(this._rawModule.actions, fn)
 }
 }

 forEachMutation (fn) {
 if (this._rawModule.mutations) {
 forEachValue(this._rawModule.mutations, fn)
 }
 }
}

```

来看一下 `Module` 的构造函数，对于每个模块而言，`this._rawModule` 表示模块的配置，`this._children` 表示它的所有子模块，`this.state` 表示这个模块定义的 `state`。

回到 `register`，那么在实例化一个 `Module` 后，判断当前的 `path` 的长度如果为 0，则说明它是一个根模块，所以把 `newModule` 赋值给了 `this.root`，否则就需要建立父子关系了：

```

const parent = this.get(path.slice(0, -1))
parent.addChild(path[path.length - 1], newModule)

```

我们先大体上了解它的逻辑：首先根据路径获取到父模块，然后再调用父模块的 `addChild` 方法建立父子关系。

`register` 的最后一步，就是遍历当前模块定义中的所有 `modules`，根据 `key` 作为 `path`，递归调用 `register` 方法，这样我们再回过头看一下建立父子关系的逻辑，首先执行了 `this.get(path.slice(0, -1))` 方法：

```
get (path) {
 return path.reduce((module, key) => {
 return module.getChild(key)
 }, this.root)
}
```

传入的 `path` 是它的父模块的 `path`，然后从根模块开始，通过 `reduce` 方法一层层去找到对应的模块，查找的过程中，执行的是 `module.getChild(key)` 方法：

```
getChild (key) {
 return this._children[key]
}
```

其实就是在返回当前模块的 `_children` 中对应 `key` 的模块，那么每个模块的 `_children` 是如何添加的呢，是通过执行 `parent.addChild(path[path.length - 1], newModule)` 方法：

```
addChild (key, module) {
 this._children[key] = module
}
```

所以说对于 `root module` 的下一层 `modules` 来说，它们的 `parent` 就是 `root module`，那么他们就会被添加到 `root module` 的 `_children` 中。每个子模块通过路径找到它的父模块，然后通过父模块的 `addChild` 方法建立父子关系，递归执行这样的过程，最终就建立一颗完整的模块树。

## 安装模块

初始化模块后，执行安装模块的相关逻辑，它的目标就是对模块中的 `state`、`getters`、`mutations`、`actions` 做初始化工作，它的入口代码是：

```
const state = this._modules.root.state
installModule(this, state, [], this._modules.root)
```

来看一下 `installModule` 的定义：

```
function installModule (store, rootState, path, module, hot) {
 const isRoot = !path.length
 const namespace = store._modules.getNamespace(path)

 // register in namespace map
```

```

if (module.namespaced) {
 store._modulesNamespaceMap[namespace] = module
}

// set state
if (!isRoot && !hot) {
 const parentState = getNestedState(rootState, path.slice(0, -1))
 const moduleName = path[path.length - 1]
 store._withCommit(() => {
 Vue.set(parentState, moduleName, module.state)
 })
}

const local = module.context = makeLocalContext(store, namespace, path)

module.forEachMutation((mutation, key) => {
 const namespacedType = namespace + key
 registerMutation(store, namespacedType, mutation, local)
})

module.forEachAction((action, key) => {
 const type = action.root ? key : namespace + key
 const handler = action.handler || action
 registerAction(store, type, handler, local)
})

module.forEachGetter((getter, key) => {
 const namespacedType = namespace + key
 registerGetter(store, namespacedType, getter, local)
})

module.forEachChild((child, key) => {
 installModule(store, rootState, path.concat(key), child, hot)
})
}

```

`installModule` 方法支持 5 个参数，`store` 表示 `root store`；`state` 表示 `root state`；`path` 表示模块的访问路径；`module` 表示当前的模块，`hot` 表示是否是热更新。

接下来看函数逻辑，这里涉及到了命名空间的概念，默认情况下，模块内部的 `action`、`mutation` 和 `getter` 是注册在全局命名空间的——这样使得多个模块能够对同一 `mutation` 或 `action` 作出响应。如果我们希望模块具有更高的封装度和复用性，可以通过添加 `namespaced: true` 的方式使其成为带命名空间的模块。当模块被注册后，它的所有 `getter`、`action` 及 `mutation` 都会自动根据模块注册的路径调整命名。例如：

```

const store = new Vuex.Store({
 modules: {
 account: {
 namespaced: true,

```

```

// 模块内容 (module assets)
state: { ... }, // 模块内的状态已经是嵌套的了，使用 `namespaced` 属性不会对其产生影响
getters: {
 isAdmin () { ... } // -> getters['account/isAdmin']
},
actions: {
 login () { ... } // -> dispatch('account/login')
},
mutations: {
 login () { ... } // -> commit('account/login')
},

// 嵌套模块
modules: {
 // 继承父模块的命名空间
 myPage: {
 state: { ... },
 getters: {
 profile () { ... } // -> getters['account/profile']
 }
 },
 // 进一步嵌套命名空间
 posts: {
 namespaced: true,
 state: { ... },
 getters: {
 popular () { ... } // -> getters['account/posts/popular']
 }
 }
}
})

```

回到 `installModule` 方法，我们首先根据 `path` 获取 `namespace`：

```
const namespace = store._modules.getNamespace(path)
```

`getNamespace` 的定义在 `src/module/module-collection.js` 中：

```

getNamespace (path) {
 let module = this.root
 return path.reduce((namespace, key) => {
 module = module.getChild(key)
 return namespace + (module.namespaced ? key + '/' : '')
 }, '')
}

```

```
}
```

从 `root module` 开始，通过 `reduce` 方法一层层找子模块，如果发现该模块配置了 `namespaced` 为 `true`，则把该模块的 `key` 拼到 `namesapce` 中，最终返回完整的 `namespace` 字符串。

回到 `installModule` 方法，接下来把 `namespace` 对应的模块保存下来，为了方便以后能根据 `namespace` 查找模块：

```
if (module.namespaced) {
 store._modulesNamespaceMap[namespace] = module
}
```

接下来判断非 `root module` 且非 `hot` 的情况执行一些逻辑，我们稍后再看。

接着是很重要的逻辑，构造了一个本地上下文环境：

```
const local = module.context = makeLocalContext(store, namespace, path)
```

来看一下 `makeLocalContext` 实现：

```
function makeLocalContext (store, namespace, path) {
 const noNamespace = namespace === ''

 const local = {
 dispatch: noNamespace ? store.dispatch : (_type, _payload, _options) => {
 const args = unifyObjectStyle(_type, _payload, _options)
 const { payload, options } = args
 let { type } = args

 if (!options || !options.root) {
 type = namespace + type
 if (process.env.NODE_ENV !== 'production' && !store._actions[type]) {
 console.error(`[vuex] unknown local action type: ${args.type}, global type: ${type}`)
 return
 }
 }
 return store.dispatch(type, payload)
 },
 commit: noNamespace ? store.commit : (_type, _payload, _options) => {
 const args = unifyObjectStyle(_type, _payload, _options)
 const { payload, options } = args
 let { type } = args

 if (!options || !options.root) {
 type = namespace + type
 }
 }
 }
}
```

```

 if (process.env.NODE_ENV !== 'production' && !store._mutations[type]) {
 console.error(`[vuex] unknown local mutation type: ${args.type}, global type: ${type}`)
 return
 }
 }

 store.commit(type, payload, options)
}
}

// getters and state object must be gotten lazily
// because they will be changed by vm update
Object.defineProperties(local, {
 getters: {
 get: noNamespace
 ? () => store.getters
 : () => makeLocalGetters(store, namespace)
 },
 state: {
 get: () => getNestedState(store.state, path)
 }
})

return local
}

```

`makeLocalContext` 支持 3 个参数相关，`store` 表示 `root store`；`namespace` 表示模块的命名空间，`path` 表示模块的 `path`。

该方法定义了 `local` 对象，对于 `dispatch` 和 `commit` 方法，如果没有 `namespace`，它们就直接指向了 `root store` 的 `dispatch` 和 `commit` 方法，否则会创建方法，把 `type` 自动拼接上 `namespace`，然后执行 `store` 上对应的方法。

对于 `getters` 而言，如果没有 `namespace`，则直接返回 `root store` 的 `getters`，否则返回 `makeLocalGetters(store, namespace)` 的返回值：

```

function makeLocalGetters (store, namespace) {
 const gettersProxy = {}

 const splitPos = namespace.length
 Object.keys(store.getters).forEach(type => {
 // skip if the target getter is not match this namespace
 if (type.slice(0, splitPos) !== namespace) return

 // extract local getter type
 const localType = type.slice(splitPos)

 // Add a port to the getters proxy.
 // Define as getter property because
 })
}
```

```
// we do not want to evaluate the getters in this time.
Object.defineProperty(gettersProxy, localType, {
 get: () => store.getters[type],
 enumerable: true
})
})

return gettersProxy
}
```

`makeLocalGetters` 首先获取了 `namespace` 的长度，然后遍历 `root store` 下的所有 `getters`，先判断它的类型是否匹配 `namespace`，只有匹配的时候我们从 `namespace` 的位置截取后面的字符串得到 `localType`，接着用 `Object.defineProperty` 定义了 `gettersProxy`，获取 `localType` 实际上是访问了 `store.getters[type]`。

回到 `makeLocalContext` 方法，再来看一下对 `state` 的实现，它的获取则是通过 `getNestedState(store.state, path)` 方法：

```
function getNestedState (state, path) {
 return path.length
 ? path.reduce((state, key) => state[key], state)
 : state
}
```

`getNestedState` 逻辑很简单，从 `root state` 开始，通过 `path.reduce` 方法一层层查找子模块 `state`，最终找到目标模块的 `state`。

那么构造完 `local` 上下文后，我们再回到 `installModule` 方法，接下来它就会遍历模块中定义的 `mutations`、`actions`、`getters`，分别执行它们的注册工作，它们的注册逻辑都大同小异。

- `registerMutation`

```
module.forEachMutation((mutation, key) => {
 const namespacedType = namespace + key
 registerMutation(store, namespacedType, mutation, local)
})

function registerMutation (store, type, handler, local) {
 const entry = store._mutations[type] || (store._mutations[type] = [])
 entry.push(function wrappedMutationHandler (payload) {
 handler.call(store, local.state, payload)
 })
}
```

首先遍历模块中的 `mutations` 的定义，拿到每一个 `mutation` 和 `key`，并把 `key` 拼接上 `namespace`，然后执行 `registerMutation` 方法。该方法实际上就是给 `root store` 上的 `_mutations[types]` 添加 `wrappedMutationHandler` 方法，该方法的具体实现我们之后会提到。注意，同一 `type` 的 `_mutations` 可以对应多个方法。

- `registerAction`

```

module.forEachAction((action, key) => {
 const type = action.root ? key : namespace + key
 const handler = action.handler || action
 registerAction(store, type, handler, local)
})

function registerAction (store, type, handler, local) {
 const entry = store._actions[type] || (store._actions[type] = [])
 entry.push(function wrappedActionHandler (payload, cb) {
 let res = handler.call(store, {
 dispatch: local.dispatch,
 commit: local.commit,
 getters: local.getters,
 state: local.state,
 rootGetters: store.getters,
 rootState: store.state
 }, payload, cb)
 if (!isPromise(res)) {
 res = Promise.resolve(res)
 }
 if (store._devtoolHook) {
 return res.catch(err => {
 store._devtoolHook.emit('vuex:error', err)
 throw err
 })
 } else {
 return res
 }
 })
}

```

首先遍历模块中的 `actions` 的定义，拿到每一个 `action` 和 `key`，并判断 `action.root`，如果否的情况把 `key` 拼接上 `namespace`，然后执行 `registerAction` 方法。该方法实际上就是给 `root store` 上的 `_actions[types]` 添加 `wrappedActionHandler` 方法，该方法的具体实现我们之后会提到。注意，同一 `type` 的 `_actions` 可以对应多个方法。

- `registerGetter`

```

module.forEachGetter((getter, key) => {
 const namespacedType = namespace + key
 registerGetter(store, namespacedType, getter, local)
})

function registerGetter (store, type, rawGetter, local) {
 if (store._wrappedGetters[type]) {
 if (process.env.NODE_ENV !== 'production') {

```

```

 console.error(`[vuex] duplicate getter key: ${type}`)
 }
 return
}
store._wrappedGetters[type] = function wrappedGetter (store) {
 return rawGetter(
 local.state, // local state
 local.getters, // local getters
 store.state, // root state
 store.getters // root getters
)
}
}
}

```

首先遍历模块中的 `getters` 的定义，拿到每一个 `getter` 和 `key`，并把 `key` 拼接上 `namespace`，然后执行 `registerGetter` 方法。该方法实际上就是给 `root store` 上的 `_wrappedGetters[key]` 指定 `wrappedGetter` 方法，该方法的具体实现我们之后会提到。注意，同一 `type` 的 `_wrappedGetters` 只能定义一个。

再回到 `installModule` 方法，最后一步就是遍历模块中的所有子 `modules`，递归执行 `installModule` 方法：

```

module.forEachChild((child, key) => {
 installModule(store, rootState, path.concat(key), child, hot)
})

```

之前我们忽略了非 `root module` 下的 `state` 初始化逻辑，现在来看一下：

```

if (!isRoot && !hot) {
 const parentState = getNestedState(rootState, path.slice(0, -1))
 const moduleName = path[path.length - 1]
 store._withCommit(() => {
 Vue.set(parentState, moduleName, module.state)
 })
}

```

之前我们提到过 `getNestedState` 方法，它是从 `root state` 开始，一层层根据模块名能访问到对应 `path` 的 `state`，那么它每一层关系的建立实际上就是通过这段 `state` 的初始化逻辑。`store._withCommit` 方法我们之后再介绍。

所以 `installModule` 实际上就是完成了模块下的 `state`、`getters`、`actions`、`mutations` 的初始化工作，并且通过递归遍历的方式，就完成了所有子模块的安装工作。

## 初始化 `store._vm`

`Store` 实例化的最后一步，就是执行初始化 `store._vm` 的逻辑，它的入口代码是：

```
resetStoreVM(this, state)
```

来看一下 `resetStoreVM` 的定义：

```
function resetStoreVM (store, state, hot) {
 const oldVm = store._vm

 // bind store public getters
 store.getters = {}
 const wrappedGetters = store._wrappedGetters
 const computed = {}
 forEachValue(wrappedGetters, (fn, key) => {
 // use computed to leverage its lazy-caching mechanism
 computed[key] = () => fn(store)
 Object.defineProperty(store.getters, key, {
 get: () => store._vm[key],
 enumerable: true // for local getters
 })
 })

 // use a Vue instance to store the state tree
 // suppress warnings just in case the user has added
 // some funky global mixins
 const silent = Vue.config.silent
 Vue.config.silent = true
 store._vm = new Vue({
 data: {
 $$state: state
 },
 computed
 })
 Vue.config.silent = silent

 // enable strict mode for new vm
 if (store.strict) {
 enableStrictMode(store)
 }

 if (oldVm) {
 if (hot) {
 // dispatch changes in all subscribed watchers
 // to force getter re-evaluation for hot reloading.
 store._withCommit(() => {
 oldVm._data.$$state = null
 })
 }
 Vue.nextTick(() => oldVm.$destroy())
 }
}
```

`resetStoreVM` 的作用实际上是想建立 `getters` 和 `state` 的联系，因为从设计上 `getters` 的获取就依赖了 `state`，并且希望它的依赖能被缓存起来，且只有当它的依赖值发生了改变才会被重新计算。因此这里利用了 Vue 中用 `computed` 计算属性来实现。

`resetStoreVM` 首先遍历了 `_wrappedGetters` 获得每个 `getter` 的函数 `fn` 和 `key`，然后定义了 `computed[key] = () => fn(store)`。我们之前提到过 `_wrappedGetters` 的初始化过程，这里 `fn(store)` 相当于执行如下方法：

```
store._wrappedGetters[type] = function wrappedGetter (store) {
 return rawGetter(
 local.state, // local state
 local.getters, // local getters
 store.state, // root state
 store.getters // root getters
)
}
```

返回的就是 `rawGetter` 的执行函数，`rawGetter` 就是用户定义的 `getter` 函数，它的前 2 个参数是 `local state` 和 `local getters`，后 2 个参数是 `root state` 和 `root getters`。

接着实例化一个 Vue 实例 `store._vm`，并把 `computed` 传入：

```
store._vm = new Vue({
 data: {
 $$state: state
 },
 computed
})
```

我们发现 `data` 选项里定义了 `$$state` 属性，而我们访问 `store.state` 的时候，实际上会访问 `store` 类上定义的 `state` 的 `get` 方法：

```
get state () {
 return this._vm._data.$$state
}
```

它实际上就访问了 `store._vm._data.$$state`。那么 `getters` 和 `state` 是如何建立依赖逻辑的呢，我们再看这段代码逻辑：

```
forEachValue(wrappedGetters, (fn, key) => {
 // use computed to leverage its lazy-caching mechanism
 computed[key] = () => fn(store)
 Object.defineProperty(store.getters, key, {
 get: () => store._vm[key],
 enumerable: true // for local getters
 })
})
```

当我根据 `key` 访问 `store.getters` 的某一个 `getter` 的时候，实际上就是访问了 `store._vm[key]`，也就是 `computed[key]`，在执行 `computed[key]` 对应的函数的时候，会执行 `rawGetter(local.state, ...)` 方法，那么就会访问到 `store.state`，进而访问到 `store._vm_data.$$state`，这样就建立了一个依赖关系。当 `store.state` 发生变化的时候，下一次再访问 `store.getters` 的时候会重新计算。

我们再来看一下 `strict mode` 的逻辑：

```
if (store.strict) {
 enableStrictMode(store)
}

function enableStrictMode (store) {
 store._vm.$watch(function () { return this._data.$$state }, () => {
 if (process.env.NODE_ENV !== 'production') {
 assert(store._committing, `Do not mutate vuex store state outside mutation handlers.`)
 }
 }, { deep: true, sync: true })
}
```

当严格模式下，`store._vm` 会添加一个 `watcher` 来观测 `this._data.$$state` 的变化，也就是当 `store.state` 被修改的时候，`store._committing` 必须为 `true`，否则在开发阶段会报警告。`store._committing` 默认值是 `false`，那么它什么时候会 `true` 呢，`Store` 定义了 `_withCommit` 实例方法：

```
_withCommit (fn) {
 const committing = this._committing
 this._committing = true
 fn()
 this._committing = committing
}
```

它就是对 `fn` 包装了一个环境，确保在 `fn` 中执行任何逻辑的时候 `this._committing = true`。所以外部任何非通过 Vuex 提供的接口直接操作修改 `state` 的行为都会在开发阶段触发警告。

## 总结

那么至此，Vuex 的初始化过程就分析完毕了，除了安装部分，我们直播重点分析了 `Store` 的实例化过程。我们要把 `store` 想象成一个数据仓库，为了更方便的管理仓库，我们把一个大的 `store` 拆成一些 `modules`，整个 `modules` 是一个树型结构。每个 `module` 又分别定义了 `state`，`getters`，`mutations`、`actions`，我们也通过递归遍历模块的方式都完成了它们的初始化。为了 `module` 具有更高的封装度和复用性，还定义了 `namespace` 的概念。最后我们还定义了一个内部的 `Vue` 实例，用来建立 `state` 到 `getters` 的联系，并且可以在严格模式下监测 `state` 的变化是不是来自外部，确保改变 `state` 的唯一途径就是显式地提交 `mutation`。

这一节我们已经建立好 `store`，接下来就是对外提供了一些 API 方便我们对这个 `store` 做数据存取的操作，下一节我们就来从源码角度来分析 `Vuex` 提供的一系列 API。

# API

上一节我们对 Vuex 的初始化过程有了深入的分析，在我们构造好这个 `store` 后，需要提供一些 API 对这个 `store` 做存取的操作，那么这一节我们就从源码的角度对这些 API 做分析。

## 数据获取

Vuex 最终存储的数据是在 `state` 上的，我们之前分析过在 `store.state` 存储的是 `root state`，那么对于模块上的 `state`，假设我们有 2 个嵌套的 `modules`，它们的 `key` 分别为 `a` 和 `b`，我们可以通过 `store.state.a.b.xxx` 的方式去获取。它的实现是在发生在 `installModule` 的时候：

```
function installModule (store, rootState, path, module, hot) {
 const isRoot = !path.length

 // ...
 // set state
 if (!isRoot && !hot) {
 const parentState = getNestedState(rootState, path.slice(0, -1))
 const moduleName = path[path.length - 1]
 store._withCommit(() => {
 Vue.set(parentState, moduleName, module.state)
 })
 }
 // ...
}
```

在递归执行 `installModule` 的过程中，就完成了整个 `state` 的建设，这样我们就可以通过 `module` 名的 `path` 去访问到一个深层 `module` 的 `state`。

有些时候，我们获取的数据不仅仅是一个 `state`，而是由多个 `state` 计算而来，Vuex 提供了 `getters`，允许我们定义一个 `getter` 函数，如下：

```
getters: {
 total (state, getters, localState, localGetters) {
 // 可访问全局 state 和 getters，以及如果是在 modules 下面，可以访问到局部 state 和 局部 getters
 return state.a + state.b
 }
}
```

我们在 `installModule` 的过程中，递归执行了所有 `getters` 定义的注册，在之后的 `resetStoreVM` 过程中，执行了 `store.getters` 的初始化工作：

```

function installModule (store, rootState, path, module, hot) {
 // ...
 const namespace = store._modules.getNamespace(path)
 // ...
 const local = module.context = makeLocalContext(store, namespace, path)

 // ...

 module.forEachGetter((getter, key) => {
 const namespacedType = namespace + key
 registerGetter(store, namespacedType, getter, local)
 })

 // ...
}

function registerGetter (store, type, rawGetter, local) {
 if (store._wrappedGetters[type]) {
 if (process.env.NODE_ENV !== 'production') {
 console.error(`[vuex] duplicate getter key: ${type}`)
 }
 return
 }
 store._wrappedGetters[type] = function wrappedGetter (store) {
 return rawGetter(
 local.state, // local state
 local.getters, // local getters
 store.state, // root state
 store.getters // root getters
)
 }
}
}

function resetStoreVM (store, state, hot) {
 // ...
 // bind store public getters
 store.getters = {}
 const wrappedGetters = store._wrappedGetters
 const computed = {}
 forEachValue(wrappedGetters, (fn, key) => {
 // use computed to leverage its lazy-caching mechanism
 computed[key] = () => fn(store)
 Object.defineProperty(store.getters, key, {
 get: () => store._vm[key],
 enumerable: true // for local getters
 })
 })

 // use a Vue instance to store the state tree
}

```

```
// suppress warnings just in case the user has added
// some funky global mixins
// ...
store._vm = new Vue({
 data: {
 $$state: state
 },
 computed
})
// ...
}
```

在 `installModule` 的过程中，为建立了每个模块的上下文环境，因此当我们访问 `store.getters.xxx` 的时候，实际上就是执行了 `rawGetter(local.state, ...)`，`rawGetter` 就是我们定义的 `getter` 方法，这也就是为什么我们的 `getter` 函数支持这四个参数，并且除了全局的 `state` 和 `getter` 外，我们还可以访问到当前 `module` 下的 `state` 和 `getter`。

## 数据存储

Vuex 对数据存储的存储本质上就是对 `state` 做修改，并且只允许我们通过提交 `mutation` 的形式去修改 `state`，`mutation` 是一个函数，如下：

```
mutations: {
 increment (state) {
 state.count++
 }
}
```

`mutations` 的初始化也是在 `installModule` 的时候：

```
function installModule (store, rootState, path, module, hot) {
 // ...
 const namespace = store._modules.getNamespace(path)

 // ...
 const local = module.context = makeLocalContext(store, namespace, path)

 module.forEachMutation((mutation, key) => {
 const namespacedType = namespace + key
 registerMutation(store, namespacedType, mutation, local)
 })
 // ...
}

function registerMutation (store, type, handler, local) {
 const entry = store._mutations[type] || (store._mutations[type] = [])
 entry.push(function wrappedMutationHandler (payload) {
 handler.call(store, local.state, payload)
 })
}
```

```

 })
}

```

`store` 提供了 `commit` 方法让我们提交一个 `mutation` :

```

commit (_type, _payload, _options) {
 // check object-style commit
 const {
 type,
 payload,
 options
 } = unifyObjectStyle(_type, _payload, _options)

 const mutation = { type, payload }
 const entry = this._mutations[type]
 if (!entry) {
 if (process.env.NODE_ENV !== 'production') {
 console.error(`[vuex] unknown mutation type: ${type}`)
 }
 return
 }
 this._withCommit(() => {
 entry.forEach(function commitIterator (handler) {
 handler(payload)
 })
 })
 this._subscribers.forEach(sub => sub(mutation, this.state))

 if (
 process.env.NODE_ENV !== 'production' &&
 options && options.silent
) {
 console.warn(
 `[vuex] mutation type: ${type}. Silent option has been removed. ` +
 'Use the filter functionality in the vue-devtools'
)
 }
}

```

这里传入的 `_type` 就是 `mutation` 的 `type`，我们可以从 `store._mutations` 找到对应的函数数组，遍历它们执行获取到每个 `handler` 然后执行，实际上就是执行了 `hwrappedMutationHandler(payload)`，接着会执行我们定义的 `mutation` 函数，并传入当前模块的 `state`，所以我们的 `mutation` 函数也就是对当前模块的 `state` 做修改。

需要注意的是，`mutation` 必须是同步函数，但是我们在开发实际项目中，经常会遇到要先去发送一个请求，然后根据请求的结果去修改 `state`，那么单纯只通过 `mutation` 是无法完成需求，因此 Vuex 又给我们设计了一个 `action` 的概念。

`action` 类似于 `mutation`，不同在于 `action` 提交的是 `mutation`，而不是直接操作 `state`，并且它可以包含任意异步操作。例如：

```
mutations: {
 increment (state) {
 state.count++
 }
},
actions: {
 increment (context) {
 setTimeout(() => {
 context.commit('increment')
 }, 0)
 }
}
```

`actions` 的初始化也是在 `installModule` 的时候：

```
function installModule (store, rootState, path, module, hot) {
 // ...
 const namespace = store._modules.getNamespace(path)

 // ...
 const local = module.context = makeLocalContext(store, namespace, path)

 module.forEachAction((action, key) => {
 const type = action.root ? key : namespace + key
 const handler = action.handler || action
 registerAction(store, type, handler, local)
 })
 // ...
}

function registerAction (store, type, handler, local) {
 const entry = store._actions[type] || (store._actions[type] = [])
 entry.push(function wrappedActionHandler (payload, cb) {
 let res = handler.call(store, {
 dispatch: local.dispatch,
 commit: local.commit,
 getters: local.getters,
 state: local.state,
 rootGetters: store.getters,
 rootState: store.state
 }, payload, cb)
 if (!isPromise(res)) {
 res = Promise.resolve(res)
 }
 if (store._devtoolHook) {
 return res.catch(err => {
```

```

 store._devtoolHook.emit('vuex:error', err)
 throw err
 })
} else {
 return res
}
})
}

```

`store` 提供了 `dispatch` 方法让我们提交一个 `action` :

```

dispatch (_type, _payload) {
 // check object-style dispatch
 const {
 type,
 payload
 } = unifyObjectStyle(_type, _payload)

 const action = { type, payload }
 const entry = this._actions[type]
 if (!entry) {
 if (process.env.NODE_ENV !== 'production') {
 console.error(`[vuex] unknown action type: ${type}`)
 }
 return
 }

 this._actionSubscribers.forEach(sub => sub(action, this.state))

 return entry.length > 1
 ? Promise.all(entry.map(handler => handler(payload)))
 : entry[0](payload)
}

```

这里传入的 `_type` 就是 `action` 的 `type`，我们可以从 `store._actions` 找到对应的函数数组，遍历它们执行获取到每个 `handler` 然后执行，实际上就是执行了 `wrappedActionHandler(payload)`，接着会执行我们定义的 `action` 函数，并传入一个对象，包含了当前模块下的 `dispatch`、`commit`、`getters`、`state`，以及全局的 `rootState` 和 `rootGetters`，所以我们定义的 `action` 函数能拿到当前模块下的 `commit` 方法。

因此 `action` 比我们自己写一个函数执行异步操作然后提交 `mutation` 的好处是在于它可以在参数中获取到当前模块的一些方法和状态，Vuex 帮我们做好了这些。

## 语法糖

我们知道 `store` 是 `Store` 对象的一个实例，它是一个原生的 Javascript 对象，我们可以在任意地方使用它们。但大部分的使用场景还是在组件中使用，那么我们之前介绍过，在 Vuex 安装阶段，它会往每一个组件实例上混入 `beforeCreate` 钩子函数，然后往组件实例上添加一个 `$store` 的实

例，它指向的就是我们实例化的 `store`，因此我们可以在组件中访问到 `store` 的任何属性和方法。

比如我们在组件中访问 `state`：

```
const Counter = {
 template: `<div>{{ count }}</div>`,
 computed: {
 count () {
 return this.$store.state.count
 }
 }
}
```

但是当一个组件需要获取多个状态时候，将这些状态都声明为计算属性会有些重复和冗余。同样这些问题也存在于 `getter`、`mutation` 和 `action`。

为了解决这个问题，Vuex 提供了一系列 `mapXXX` 辅助函数帮助我们实现在组件中可以很方便的注入 `store` 的属性和方法。

## mapState

我们先来看一下 `mapState` 的用法：

```
// 在单独构建的版本中辅助函数为 Vuex.mapState
import { mapState } from 'vuex'

export default {
 // ...
 computed: mapState({
 // 箭头函数可使代码更简练
 count: state => state.count,

 // 传字符串参数 'count' 等同于 `state => state.count`
 countAlias: 'count',

 // 为了能够使用 `this` 获取局部状态，必须使用常规函数
 countPlusLocalState (state) {
 return state.count + this.localCount
 }
 })
}
```

再来看一下 `mapState` 方法的定义，在 `src/helpers.js` 中：

```
export const mapState = normalizeNamespace((namespace, states) => {
 const res = []
 normalizeMap(states).forEach(({ key, val }) => {
```

```

res[key] = function mappedState () {
 let state = this.$store.state
 let getters = this.$store.getters
 if (namespace) {
 const module = getModuleByNamespace(this.$store, 'mapState', namespace)
 if (!module) {
 return
 }
 state = module.context.state
 getters = module.context.getters
 }
 return typeof val === 'function'
 ? val.call(this, state, getters)
 : state[val]
}
// mark vuex getter for devtools
res[key].vuex = true
})
return res
})

function normalizeNamespace (fn) {
 return (namespace, map) => {
 if (typeof namespace !== 'string') {
 map = namespace
 namespace = ''
 } else if (namespace.charAt(namespace.length - 1) !== '/') {
 namespace += '/'
 }
 return fn(namespace, map)
 }
}

function normalizeMap (map) {
 return Array.isArray(map)
 ? map.map(key => ({ key, val: key }))
 : Object.keys(map).map(key => ({ key, val: map[key] }))
}

```

首先 `mapState` 是通过执行 `normalizeNamespace` 返回的函数，它接收 2 个参数，其中 `namespace` 表示命名空间，`map` 表示具体的对象，`namespace` 可不传，稍后我们来介绍 `namespace` 的作用。

当执行 `mapState(map)` 函数的时候，实际上就是执行 `normalizeNamespace` 包裹的函数，然后把 `map` 作为参数 `states` 传入。

`mapState` 最终是要构造一个对象，每个对象的元素都是一个方法，因为这个对象是要扩展到组件的 `computed` 计算属性中的。函数首先执行 `normalizeMap` 方法，把这个 `states` 变成一个数组，数组的每个元素都是 `{key, val}` 的形式。接着再遍历这个数组，以 `key` 作为对象的 `key`，值为一

个 `mappedState` 的函数，在这个函数的内部，获取到 `$store.getters` 和 `$store.state`，然后再判断数组的 `val` 如果是一个函数，执行该函数，传入 `state` 和 `getters`，否则直接访问 `state[val]`。

比起一个个手动声明计算属性，`mapState` 确实要方便许多，下面我们来看一下 `namespace` 的作用。

当我们想访问一个子模块的 `state` 的时候，我们可能需要这样访问：

```
computed: {
 mapState({
 a: state => state.some.nested.module.a,
 b: state => state.some.nested.module.b
 })
},
```

这样从写法上就很不友好，`mapState` 支持传入 `namespace`，因此我们可以这么写：

```
computed: {
 mapState('some/nested/module', {
 a: state => state.a,
 b: state => state.b
 })
},
```

这样看起来就清爽许多。在 `mapState` 的实现中，如果有 `namespace`，则尝试去通过 `getModuleByNamespace(this.$store, 'mapState', namespace)` 对应的 `module`，然后把 `state` 和 `getters` 修改为 `module` 对应的 `state` 和 `getters`。

```
function getModuleByNamespace (store, helper, namespace) {
 const module = store._modulesNamespaceMap[namespace]
 if (process.env.NODE_ENV !== 'production' && !module) {
 console.error(`[vuex] module namespace not found in ${helper}(): ${namespace}`)
 }
 return module
}
```

我们在 Vuex 初始化执行 `installModule` 的过程中，初始化了这个映射表：

```
function installModule (store, rootState, path, module, hot) {
 // ...
 const namespace = store._modules.getNamespace(path)

 // register in namespace map
 if (module.namespaced) {
 store._modulesNamespaceMap[namespace] = module
 }
}
```

```
// ...
}
```

## mapGetters

我们先来看一下 `mapGetters` 的用法：

```
import { mapGetters } from 'vuex'

export default {
 // ...
 computed: {
 // 使用对象展开运算符将 getter 混入 computed 对象中
 ...mapGetters([
 'doneTodosCount',
 'anotherGetter',
 // ...
])
 }
}
```

和 `mapState` 类似，`mapGetters` 是将 `store` 中的 `getter` 映射到局部计算属性，来看一下它的定义：

```
export const mapGetters = normalizeNamespace((namespace, getters) => {
 const res = {}
 normalizeMap(getters).forEach(({ key, val }) => {
 // the namespace has been mutate by normalizeNamespace
 val = namespace + val
 res[key] = function mappedGetter () {
 if (namespace && !getModuleByNamespace(this.$store, 'mapGetters', namespace)) {
 return
 }
 if (process.env.NODE_ENV !== 'production' && !(val in this.$store.getters)) {
 console.error(`[vuex] unknown getter: ${val}`)
 return
 }
 return this.$store.getters[val]
 }
 // mark vuex getter for devtools
 res[key].vuex = true
 })
 return res
})
```

`mapGetters` 也同样支持 `namespace`，如果不写 `namespace`，访问一个子 `module` 的属性需要写很长的 `key`，一旦我们使用了 `namespace`，就可以方便我们的书写，每个 `mappedGetter` 的实现实际上就是取 `this.$store.getters[val]`。

## mapMutations

我们可以在组件中使用 `this.$store.commit('xxx')` 提交 `mutation`，或者使用 `mapMutations` 辅助函数将组件中的 `methods` 映射为 `store.commit` 的调用。

我们先来看一下 `mapMutations` 的用法：

```
import { mapMutations } from 'vuex'

export default {
 // ...
 methods: {
 ...mapMutations([
 'increment', // 将 `this.increment()` 映射为 `this.$store.commit('increment')`

 // `mapMutations` 也支持载荷：
 'incrementBy' // 将 `this.incrementBy(amount)` 映射为 `this.$store.commit('incrementBy', amount)`
]),
 ...mapMutations({
 add: 'increment' // 将 `this.add()` 映射为 `this.$store.commit('increment')`
 })
 }
}
```

`mapMutations` 支持传入一个数组或者一个对象，目标都是组件中对应的 `methods` 映射为 `store.commit` 的调用。来看一下它的定义：

```
export const mapMutations = normalizeNamespace((namespace, mutations) => {
 const res = []
 normalizeMap(mutations).forEach(({ key, val }) => {
 res[key] = function mappedMutation (...args) {
 // Get the commit method from store
 let commit = this.$store.commit
 if (namespace) {
 const module = getModuleByNamespace(this.$store, 'mapMutations', namespace)
 if (!module) {
 return
 }
 commit = module.context.commit
 }
 return typeof val === 'function'
 ? val.apply(this, [commit].concat(args))
 : commit.apply(this.$store, [val].concat(args))
 }
 })
})
```

```

 })
 return res
})

```

可以看到 `mappedMutation` 同样支持了 `namespace`，并且支持了传入额外的参数 `args`，作为提交 `mutation` 的 `payload`，最终就是执行了 `store.commit` 方法，并且这个 `commit` 会根据传入的 `namespace` 映射到对应 `module` 的 `commit` 上。

## mapActions

我们可以在组件中使用 `this.$store.dispatch('xxx')` 提交 `action`，或者使用 `mapActions` 辅助函数将组件中的 `methods` 映射为 `store.dispatch` 的调用。

`mapActions` 在用法上和 `mapMutations` 几乎一样，实现也很类似：

```

export const mapActions = normalizeNamespace((namespace, actions) => {
 const res = {}
 normalizeMap(actions).forEach(({ key, val }) => {
 res[key] = function mappedAction (...args) {
 // get dispatch function from store
 let dispatch = this.$store.dispatch
 if (namespace) {
 const module = getModuleByNamespace(this.$store, 'mapActions', namespace)
 if (!module) {
 return
 }
 dispatch = module.context.dispatch
 }
 return typeof val === 'function'
 ? val.apply(this, [dispatch].concat(args))
 : dispatch.apply(this.$store, [val].concat(args))
 }
 })
 return res
})

```

和 `mapMutations` 的实现几乎一样，不同的是把 `commit` 方法换成了 `dispatch`。

## 动态更新模块

在 Vuex 初始化阶段我们构造了模块树，初始化了模块上各个部分。在有一些场景下，我们需要动态去注入一些新的模块，Vuex 提供了模块动态注册功能，在 `store` 上提供了一个 `registerModule` 的 API。

```

registerModule (path, rawModule, options = {}) {
 if (typeof path === 'string') path = [path]

```

```

if (process.env.NODE_ENV !== 'production') {
 assert(Array.isArray(path), `module path must be a string or an Array.`)
 assert(path.length > 0, 'cannot register the root module by using registerModule.')
}

this._modules.register(path, rawModule)
installModule(this, this.state, path, this._modules.get(path), options.preserveState)
// reset store to update getters...
resetStoreVM(this, this.state)
}

```

`registerModule` 支持传入一个 `path` 模块路径和 `rawModule` 模块定义，首先执行 `register` 方法扩展我们的模块树，接着执行 `installModule` 去安装模块，最后执行 `resetStoreVM` 重新实例化 `store._vm`，并销毁旧的 `store._vm`。

相对的，有动态注册模块的需求就有动态卸载模块的需求，Vuex 提供了模块动态卸载功能，在 `store` 上提供了一个 `unregisterModule` 的 API。

```

unregisterModule (path) {
 if (typeof path === 'string') path = [path]

 if (process.env.NODE_ENV !== 'production') {
 assert(Array.isArray(path), `module path must be a string or an Array.`)
 }

 this._modules.unregister(path)
 this._withCommit(() => {
 const parentState = getNestedState(this.state, path.slice(0, -1))
 Vue.delete(parentState, path[path.length - 1])
 })
 resetStore(this)
}

```

`unregisterModule` 支持传入一个 `path` 模块路径，首先执行 `unregister` 方法去修剪我们的模块树：

```

unregister (path) {
 const parent = this.get(path.slice(0, -1))
 const key = path[path.length - 1]
 if (!parent.getChild(key).runtime) return

 parent.removeChild(key)
}

```

注意，这里只会移除我们运行时动态创建的模块。

接着会删除 `state` 在该路径下的引用，最后执行 `resetStore` 方法：

```
function resetStore (store, hot) {
 store._actions = Object.create(null)
 store._mutations = Object.create(null)
 store._wrappedGetters = Object.create(null)
 store._modulesNamespaceMap = Object.create(null)
 const state = store.state
 // init all modules
 installModule(store, state, [], store._modules.root, true)
 // reset vm
 resetStoreVM(store, state, hot)
}
```

该方法就是把 `store` 下的对应存储的 `_actions`、`_mutations`、`_wrappedGetters` 和 `_modulesNamespaceMap` 都清空，然后重新执行 `installModule` 安装所有模块以及 `resetStoreVM` 重置 `store._vm`。

## 总结

那么至此，Vuex 提供的一些常用 API 我们就分析完了，包括数据的存取、语法糖、模块的动态更新等。要理解 Vuex 提供这些 API 都是方便我们在对 `store` 做各种操作来完成各种能力，尤其是 `mapXXX` 的设计，让我们在使用 API 的时候更加方便，这也是我们今后在设计一些 JavaScript 库的时候，从 API 设计角度中应该学习的方向。

# 插件

Vuex 除了提供的存取能力，还提供了一种插件能力，让我们可以监控 `store` 的变化过程来做一些事情。

Vuex 的 `store` 接受 `plugins` 选项，我们在实例化 `Store` 的时候可以传入插件，它是一个数组，然后在执行 `Store` 构造函数的时候，会执行这些插件：

```
const {
 plugins = [],
 strict = false
} = options
// apply plugins
plugins.forEach(plugin => plugin(this))
```

在我们实际项目中，我们用到的最多的就是 Vuex 内置的 `Logger` 插件，它能够帮我们追踪 `state` 变化，然后输出一些格式化日志。下面我们就来分析这个插件的实现。

## Logger 插件

`Logger` 插件的定义在 `src/plugins/logger.js` 中：

```
export default function createLogger ({
 collapsed = true,
 filter = (mutation, stateBefore, stateAfter) => true,
 transformer = state => state,
 mutationTransformer = mut => mut,
 logger = console
} = {}) {
 return store => {
 let prevState = deepCopy(store.state)

 store.subscribe((mutation, state) => {
 if (typeof logger === 'undefined') {
 return
 }
 const nextState = deepCopy(state)

 if (filter(mutation, prevState, nextState)) {
 const time = new Date()
 const formattedTime = ` @ ${pad(time.getHours(), 2)}:${pad(time.getMinutes(), 2)}:${pad(time.getSeconds(), 2)}.${pad(time.getMilliseconds(), 3)}`
 const formattedMutation = mutationTransformer(mutation)
 const message = `mutation ${mutation.type}${formattedTime}`
 const startMessage = collapsed
 ? logger.groupCollapsed
 : logger.group(`mutation ${mutation.type}`)
```

```

 : logger.group

 // render
 try {
 startMessage.call(logger, message)
 } catch (e) {
 console.log(message)
 }

 logger.log('%c prev state', 'color: #9E9E9E; font-weight: bold', transforme
r(prevState))
 logger.log('%c mutation', 'color: #03A9F4; font-weight: bold', formattedMut
ation)
 logger.log('%c next state', 'color: #4CAF50; font-weight: bold', transforme
r(nextState))

 try {
 logger.groupEnd()
 } catch (e) {
 logger.log('— log end —')
 }
 }

 prevState = nextState
})
}
}

function repeat (str, times) {
 return (new Array(times + 1)).join(str)
}

function pad (num, maxLength) {
 return repeat('0', maxLength - num.toString().length) + num
}

```

插件函数接收的参数是 `store` 实例，它执行了 `store.subscribe` 方法，先来看一下 `subscribe` 的定义：

```

subscribe (fn) {
 return genericSubscribe(fn, this._subscribers)
}

function genericSubscribe (fn, subs) {
 if (subs.indexOf(fn) < 0) {
 subs.push(fn)
 }
 return () => {
 const i = subs.indexOf(fn)
 if (i > -1) {

```

```

 subs.splice(i, 1)
 }
}
}

```

`subscribe` 的逻辑很简单，就是往 `this._subscribers` 去添加一个函数，并返回一个 `unsubscribe` 的方法。

而我们在执行 `store.commit` 的方法的时候，会遍历 `this._` 执行它们对应的回调函数：

```

commit (_type, _payload, _options) {
 const {
 type,
 payload,
 options
 } = unifyObjectStyle(_type, _payload, _options)

 const mutation = { type, payload }
 // ...
 this._subscribers.forEach(sub => sub(mutation, this.state))
}

```

回到我们的 `Logger` 函数，它相当于订阅了 `mutation` 的提交，它的 `prevState` 表示之前的 `state`，`nextState` 表示提交 `mutation` 后的 `state`，这两个 `state` 都需要执行 `deepCopy` 方法拷贝一份对象的副本，这样对他们的修改就不会影响原始 `store.state`。

接下来就构造一些格式化的消息，打印出一些时间消息 `message`，之前的状态 `prevState`，对应的 `mutation` 操作 `formattedMutation` 以及下一个状态 `nextState`。

最后更新 `prevState = nextState`，为下一次提交 `mutation` 输出日志做准备。

## 总结

那么至此 Vuex 的插件分析就结束了，Vuex 从设计上支持了插件，让我们很好地从外部追踪 `store` 内部的变化，`Logger` 插件在我们的开发阶段也提供了很好地指引作用。当然我们也可以自己去实现 `vuex` 的插件，来帮助我们实现一些特定的需求。