Vue 高级进阶82道面试题（https://github.com/minsion）

面试题 1. 简述 v-if 和 v-for 为什么不建议一起使用？

推荐指数：★★★ 试题难度：初级 试题类型：原理题

试题回答参考思路：

v-for和v-if不要在同一标签中使用，因为解析时先解析v-for在解析v-if。如果遇到需要同时使用时可以考虑写成计算属性的方式。
永远不要把 v-if 和 v-for 同时用在同一个元素上，带来性能方面的浪费（每次渲染都会先循环再进行条件判断）
如果避免出现这种情况，则在外层嵌套template（页面渲染不生成dom节点），在这一层进行v-if判断，然后在内部进行v-for循环

如果条件出现在循环内部，可通过计算属性computed提前过滤掉那些不需要显示的项

```
computed: {
items: function() {
return this.list.filter(function (item) {
return item.isShow
})
}
}
```

面试题 2. 请解释Vue的父子组件生命周期钩子函数执行顺序？

推荐指数：★★★★ 试题难度：初级 试题类型：原理题

试题回答参考思路：

加载渲染过程
父 beforeCreate -> 父 created -> 父 beforeMount -> 子 beforeCreate -> 子 created
-> 子 beforeMount -> 子 mounted -> 父 mounted
子组件更新过程
父 beforeUpdate -> 子 beforeUpdate -> 子 updated -> 父 updated
父组件更新过程
父 beforeUpdate -> 父 updated
销毁过程 父 beforeDestroy -> 子 beforeDestroy -> 子 destroyed -> 父 destroyed
总结： 父组件先开始 子组件先结束

面试题 3. 简述对于Vue的diff算法理解？

推荐指数：★★★★★ 试题难度：高难 试题类型：原理题

试题回答参考思路：

1）diff算法的作用：用来修改dom的一小段，不会引起dom树的重绘

2）diff算法的实现原理：diff算法将virtual dom的某个节点数据改变后生成的新的vnode与旧节点进行比较，并替换为新的节点，具体过程就是调用patch方法，比较新旧节点，一边比较一边给真实的dom打补丁进行替换

3）具体过程详解：

a、在采用diff算法进行新旧节点进行比较的时候，比较是按照在同级进行比较的，不会进行跨级比较：

b、当数据发生改变的时候，set方法会调用dep.notify通知所有的订阅者watcher，订阅者会调用patch函数给响应的dom进行打补丁，从而更新真实的视图

c、patch函数接受两个参数，第一个是旧节点，第二个是新节点，首先判断两个节点是否值得比较，值得比较则执行patchVnode函数，不值得比较则直接将旧节点替换为新节点。如果两个节点一样就直接检查对应的子节点，如果子节点不一样就说明整个子节点全部改变不再往下对比直接进行新旧节点的整体替换

d、patchVnode函数：找到真实的dom元素；判断新旧节点是否指向同一个对象，如果是就直接返回；如果新旧节点都有文本节点，那么直接将新的文本节点赋值给dom元素并且更新旧的节点为新的节点；如果旧节点有子节点而新节点没有，则直接删除dom元素中的子节点；如果旧节点没有子节点，新节点有子节点，那么直接将新节点中的子节点更新到dom中；如果两者都有子节点，那么继续调用函数updateChildren

e、updateChildren函数：抽离出新旧节点的所有子节点，并且设置新旧节点的开始指针和结束指针，然后进行两辆比较，从而更新dom（调整顺序或者插入新的内容 结束后删掉多余的内容）

面试题 4. 简述vue.mixin的使用场景和原理?

推荐指数：★★★★★ 试题难度： 中级 试题类型： 原理题

试题回答参考思路：

Mixin是面向对象程序设计语言中的类，提供了方法的实现。
其他类可以访问mixin类的方法而不必成为其子类
当一段代码非常相似的时候就可以抽离成一个mixin
mixins是一个js对象，它可以包含我们组件中script项中的任意功能选项，如data、components、methods 、created、computed等等。只要将共用的功能以对象的方式传入 mixins选项中，当组件使用 mixins对象时所有mixins对象的选项都将被混入该组件本身的选项中来，这样就可以提高代码的重用性，使你的代码保持干净和易于维护。
使用场景
当存在多个组件中的数据或者功能很相近时，就可以利用mixins将公共部分提取出来，通过mixins封装的函数，组件调用他们是不会改变函数作用域外部的。

mixins和vuex的区别
vuex公共状态管理，在一个组件被引入后，如果该组件改变了vuex里面的数据状态，其他引入vuex数据的组件也会对应修改，所有的vue组件应用的都z是同一份vuex数据。（在js中，有点类似于浅拷贝）
vue引入mixins数据，mixins数据或方法，在每一个组件中都是独立的，互不干扰的，都属于vue组件自身。（在js中，有点类似于深度拷贝）
mixins和组件的区别
组件：在父组件中引入组件，相当于在父组件中给出一片独立的空间供子组件使用，然后根据props来传值，但本质上两者是相对独立的。
Mixins：则是在引入组件之后与组件中的对象和方法进行合并，相当于扩展了父组件的对象与方法，可以理解为形成了一个新的组件。
（装饰器模式）
mixin的使用
定义一个mixin名字为myMixins

```
export default {
data () {
return {
num:1
}
}
,
methods: {
mymixin() {
console.log(this.num);
}
,
}
}
```

在组件中使用

```
import {
myMixins
}
from './myMixins';
export default {
mixins: [myMixins],
data() {
return {
}
}
,
created() {
//使用mixin可以直接用，但是组件就得传值
this.num++
}
,
}
```

面试题 5. 简述vue-router 组件复用导致路由参数失效怎么办?

推荐指数: ★★★ 试题难度: 初级 试题类型: 原理题

试题回答参考思路:

解决方案 :
通 过 watch 监听 路由参数再发请求
watch: {
"router":function(){
this.getData(this.$router.params.xxx)
}
}

面试题 6. 请简述Vue3.x 响应式数据原理是什么? （ 重点 ）

推荐指数: ★★★★★ 试题难度: 高难 试题类型: 原理题

试题回答参考思路:

Vuejs 作为在众多 MVVM(Model-View-ViewModel) 框架中脱颖而出的佼佼者，无疑是值得任何一个前端开发者去深度学习的。
不可置否尤大佬的 VueJs 中有许多值得我们深入研究的内容，但是作为最核心的数据响应式 Reactivity 模块正是我们日常工作中高端相关的内容同时也是 VueJs 中最核心的内容之一。
至于 Vuejs 中的响应式原理究竟有多重要，这里我就不必累赘了。相信大家都能理解它的重要性。
不过这里我想强调的是，所谓响应式原理本质上也是基于 Js 代码的升华实现而已。你也许会觉得它很难，但是这一切只是源于你对他的未知。毕竟只要是你熟悉的 JavaScript ，那么问题就不会很大对吧。
今天我们就让我们基于最新版 Vuejs 3.2 来稍微聊聊 VueJs 中核心模块 Reactive 是如何实现数据响应式的。
前置知识
ES6 Proxy & Reflect
Proxy 是 ES6 提供给我们对于原始对象进行劫持的 Api ，同样 Reflect 内置 Api 为我们提供了对于原始对象的拦截操作。
这里我们主要是用到他们的 get 、 set 陷阱。
Typescript
TypeScript 的作用不言而喻了，文中代码我会使用 TypeScript 来书写。
Esbuild
EsBuild 是一款新型 bundle build tools ，它内部使用 Go 对于我们的代码进行打包整合。
Pnpm
pnpm 是一款优秀的包管理工具，这里我们主要用它来实现 monorepo 。
如果你还没在你的电脑上安装过 pnpm ，那么请你跟随官网安装它很简单，只需要一行 npm install -g pnpm即可。
搭建环境
工欲善其事，必先利其器。在开始之前我们首先会构建一个简陋的开发环境，便于将我们的

TypeScript 构建成为 life 形式，提供给浏览器中直接使用。
因为文章主要针对于响应式部分内容进行梳理，构建环境并不是我们的重点。所以我并不会深入构建环境的搭建中为大家讲解这些细节。
如果你有兴趣，可以跟着我一起来搭建这个简单的组织结构。如果你并不想动手，没关系。我们的重点会放在在之后的代码。
初始化项目目录
首先我们创建一个简单的文件夹，命名为 vue 执行 pnpm init -y 初始化 package.json 。
接下来我们依次创建：
pnpm-workspace.yaml文件
这是一个有关 pnpm 实现 monorepo 的 yaml 配置文件，我们会在稍微填充它。
.npmrc文件
这是有关 npm 的配置信息存放文件。
packages/reactivity目录
我们会在这个目录下实现核心的响应式原理代码，上边我们提过 vue3 目录架构基于 monorepo 的结构，所以这是一个独立用于维护响应式相关的模块目录。
当然，每个 packages 下的内容可以看作一个独立的项目，所以它们我们在 reactivity 目录中执行 pnpm init -y 初始化自己的 package.json。
同样新建 packages/reactivity/src 作为 reactivity 模块下的文件源代码。
packages/share目录
同样，正如它的文件夹名称，这个目录下存放所有 vuejs 下的工具方法，分享给别的模块进行引入使用。
它需要和 reactivity 维护相同的目录结构。
scripts/build.js文件
我们需要额外新建一个 scripts 文件夹，同时新建 scripts/build.js 用于存放构建时候的脚本文件。
image.png
此时目录如图所示。
安装依赖
接下来我们来依次安装需要使用到的依赖环境，在开始安装依赖之前。我们先来填充对应的 .npmrc 文件：
shamefully-hoist = true
默认情况下 pnpm 安装的依赖是会解决幽灵依赖的问题，所谓什么是幽灵依赖你可以查看这篇文章。
这里我们配置 shamefully-hoist = true 意为我们需要第三方包中的依赖提升，也就是需要所谓的幽灵依赖。
这是因为我们会在之后引入源生 Vue 对比实现效果与它是否一致。
你可以在这里详细看到它的含义。
同时，接下里让我们在 pnpm-workspace.yaml 来填入以下代码：
```
packages:
# 所有在 packages/ 和 components/ 子目录下的 package
- 'packages/**'
# - 'components/**'
# 不包括在 test 文件夹下的 package
# - '!**/test/**'
```
因为基于 monorepo 的方式来组织包代码，所以我们需要告诉 pnpm 我们的 repo 工作目录。

这里我们指定了 packages/ 为 monorepo 工作目录，此时我们的 packages 下的每一个文件夹都会被 pnpm 认为是一个独立的项目。
接下来我们去安装所需要的依赖：
pnpm install -D typescript vue esbuild minimist -w
注意，这里 -w 意为 --workspace-root ，表示我们将依赖安装在顶层目录，所以包可以共享到这些依赖。
同时 minimist 是 node-optimist 的核心解析模块，它的主要作为即为解析执行 Node 脚本时的环境变量。
填充构建
接下来我们就来填充构建部分逻辑。
更改 package.json
首先，让我们切换到项目跟目录下对于整个 repo 的 pacakge.json 进行改造。 {
"name": "@vue",
"version": "1.0.0",
"description": "",
"main": "index.js",
"scripts": {
"dev": "node ./scripts/dev.js reactivity -f global"
}
,
"keywords": [],
"author": "",
"license": "ISC",
"devDependencies": {
"esbuild": "^0.14.27",
"typescript": "^4.6.2",
"vue": "^3.2.31"
}
}
首先我们将包名称修改为作用域，@vue 表示该包是一个组织包。
其次，我们修改 scripts 脚本。表示当运行 pnpm run dev 时会执行 ./scripts/dev.js 同时传入一个 reactivity 参数以及 -f 的 global 环境变量。
更改项目内 package.json
接下来我们需要更改每个 repop 内的 package.json（以下简称 pck） 。这里我们以 reactivity 模块为例，share 我就不重复讲解了。 {
"name": "@vue/reactive",
"version": "1.0.0",
"description": "",
"main": "index.js",
"buildOptions": {
"name": "VueReactivity",
"formats": [
"esm-bundler",
"esm-browser",
"cjs",
"global"
]

```
}
,
"keywords": [],
"author": "",
"license": "ISC"
}
```

首先，我们将 reactivity 包中的名称改为作用域名 @vue/reactive 。

其次我们为 pck 中添加了一些自定义配置，分别为：

buildOptions.name 该选项表示打包生成 IIFE 时，该模块挂载在全局下的变量名。

buildOptions.formats 该选项表示该模块打包时候需要输出的模块规范。

填充scripts/dev.js

之后，让我们切换到 scripts/dev.js 来实现打包逻辑：

```
// scripts/dev.js
const {
build
}
= require('esbuild');
const {
resolve
}
= require('path');
const argv = require('minimist')(process.argv.slice(2));
// 获取参数 minimist
const target = argv['_'];
const format = argv['f'];
const pkg = require(resolve(__dirname, '../packages/reactivity/package.json'));
const outputFormat = format.startsWith('global')
? 'iife'
: format.startsWith('cjs')
? 'cjs'
: 'esm';
// 打包输出文件
const outfile = resolve(
__dirname,
`../packages/$ {
target
}
/dist/$ {
target
}
.$ {
outputFormat
}
.js`
);
// 调用ESbuild的NodeApi执行打包
build( {
```

```
entryPoints: [resolve(__dirname, `../packages/${target}/src/index.ts`)],
outfile,
bundle: true, // 将所有依赖打包进入
sourcemap: true, // 是否需要sourceMap
format: outputFormat, // 输出文件格式 IIFE、CJS、ESM
globalName: pkg.buildOptions?.name, // 打包后全局注册的变量名 IIFE下生效
platform: outputFormat === 'cjs' ? 'node' : 'browser', // 平台
watch: true, // 表示检测文件变动重新打包
}
);
```

脚本中的已经进行了详细的注释，这里我稍微在啰嗦一些。

其次整个流程看来像是这样，首先当我们运行 npm run dev 时，相当于执行了 node ./scripts/dev.js reactivity -f global。

所以在执行对应 dev.js 时，我们通过 minimist 获得对应的环境变量 target 和 format 表示我们本次打包分别需要打包的 package 和模式，当然你也可以通过 process.argv 自己截取。

之后我们通过判断如果传入的 -f 为 global 时将它变成 iife 模式，执行 esbuild 的 Node Api 进行打包对应的模块。

需要注意的是，ESbuild 默认支持 typescript 所以不需要任何额外处理。

当然，我们此时并没有在每个包中创建对应的入口文件。让我们分别创建两个packages/reactivity/src/index.ts以及packages/share/src/index.ts作为入口文件。

此时，当你运行 npm run dev 时，会发现会生成打包后的js文件：

image.png

写在环境结尾的话

至此，针对于一个简易版 Vuejs 的项目构建流程我们已经初步实现了。如果有兴趣深入了解这个完整流程的同学可以自行查看对应 源码。

当然这种根据环境变量进行动态打包的思想，我在之前的React-Webpack5-TypeScript打造工程化多页面应用中详细讲解过这一思路，有兴趣的同学可以自行查阅。

其实关于构建思路我大可不必在这里展开，直接讲述响应式部分代码即可。但是这一流程在我的日常工作中的确帮助过我在多页面应用业务上进行了项目构建优化。

所以我觉得还是有必要拿出来和大家稍微聊一聊这一过程，希望大家在以后业务中遇到该类场景下可以结合 Vuejs 的构建思路来设计你的项目构建流程。

响应式原理

上边我们对于构建稍稍花费了一些篇幅，接下来终于我们要步入正题进行响应式原理部分了。

首先，在开始之前我会稍微强调一些。文章中的代码并不是一比一对照源码来实现响应式原理，但是实现思想以及实现过程是和源码没有出入的。

这是因为源码中拥有非常多的条件分支判断和错误处理，同时源码中也考虑了数组、Set、Map 之类的数据结构。

这里，我们仅仅先考虑基础的对象，至于其他数据类型我会在之后的文章中详细和大家一一道来。

同时我也会在每个步骤的结尾贴出对应的源代码地址，提供给大家参照源码进行对比阅读。

开始之前

在我们开始响应式原理之前，我想和大家稍微阐述下对应背景。因为可能有部分同学对应

Vue3 中的源码并不是很了解。
在 VueJs 中的存在一个核心的 Api Effect ，这个 Api 在 Vue 3.2 版本之后暴露给了开发者去调用，在3.2之前都是 Vuejs 内部方法并不提供给开发者使用。
简单来说我们所有模版（组件）最终都会被 effect 包裹 ，当数据发生变化时 Effect 会重新执行，所以 vuejs 中的响应式原理可以说是基于 effect 来实现的 。
当然这里你仅仅需要了解，最终组件是会编译成为一个个 effect ，当响应式数据改变时会触发 effect 函数重新执行从而更新渲染页面即可。
之后我们也会详细介绍 effect 和 响应式是如何关联到一起的。
基础目录结构
首先我们来创建一些基础的目录结构：
reactivity/src/index.ts 用于统一引入导出各个模块
reactivity/src/reactivity.ts 用于维护 reactive 相关 Api。
reactivity/src/effect.ts 用户维护 effect 相关 Api。
这一步我们首先在 reactivity 中新建对应的文件：
image.png
reactive 基础逻辑处理
接下来我们首先进入相关的 reactive.ts 中去。
思路梳理
关于 Vuejs 是如何实现数据响应式，简单来说它内部利用了 Proxy Api 进行了访问/设置数据时进行了劫持。
对于数据访问时，需要进行依赖收集。记录当前数据中依赖了哪些 Effect ，当进行数据修改时候同样会进行触发更新，重新执行当前数据依赖的 Effect。简单来说，这就是所谓的响应式原理。
关于 Effect 你可以暂时的将它理解成为一个函数，当数据改变函数（Effect）重新执行从而函数执行导致页面重新渲染。
Target 实现目标
在开始书写代码之前，我们先来看看它的用法。我们先来看看 reactive 方法究竟是如何搭配 effect 进行页面的更新：

不太了解 Effect 和响应式数据的同学可以将这段代码放在浏览器下执行试试看。
首先我们使用 reactive Api 创建了一个响应式数据 reactiveData 。
之后，我们创建了一个 effect，它会接受一个 fn 作为参数 。这个 effect 内部的逻辑非常简单：它将 id 为 app 元素的内容置为 reactiveData.name 的值。
注意，这个 effect 传入的 fn 中依赖了响应式数据 reactiveData 的 name 属性，这一步

通常成为依赖收集。
当 effect 被创建时，fn 会被立即执行所以 app 元素会渲染对应的 19Qingfeng 。
当 0.5s 后 timer 达到时间，我们修改了 reactiveData 响应式数据的 name 属性，此时会触发改属性依赖的 effct 重新执行，这一步同样通常被称为触发更新。
所以页面上看起来的结果就是首先渲染出 19Qingfeng 在 0.5s 后由于响应式数据的改变导致 effect 重新执行所以修改了 app 的 innerHTML 导致页面重新渲染。
这就是一个非常简单且典型的响应式数据 Demo ，之后我们会一步一步基于结果来逆推实现这个逻辑。
基础 Reactive 方法实现
接下来我们先来实现一个基础版的 Reactive 方法，具体使用 API 你可以参照 这里。
上边我们提到过 VueJs 中针对于响应式数据本质上就是基于 Proxy & Reflect 对于数据的劫持，那么自然我们会想到这样的代码：

```
// reactivity/src/reactivity.ts
export function isPlainObj(value: any): value is object {
return typeof value === 'object' && value !== null;
}
const reactive = (obj) => {
// 传入非对象
if (!isPlainObj(obj)) {
return obj;
}
// 声明响应式数据
const proxy = new Proxy(obj, {
get() {
// dosomething
}
,
set() {
// dosomething
}
}
);
return proxy;
}
;
```

上边的代码非常简单，我们创建了一个 reactive 对象，它接受传入一个 Object 类型的对象。
我们会对于函数传入的 obj 进行校验，如果传入的是 object 类型那么会直接返回。
接下来，我们会根据传入的对象 obj 创建一个 proxy 代理对象。并且会在该代理对象上针对于 get 陷阱（访问对象属性时）以及 set （修改代理对象的值时）进行劫持从而实现一系列逻辑。
依赖收集
之前我们提到过针对于 reactive 的响应式数据会在触发 get 陷阱时会进行依赖收集。
这里你可以简单将依赖收集理解为记录当前数据被哪些Effect使用到，之后我们会一步一步来实现它。

```
// reactivity/src/reactivity.ts
export function isPlainObj(value: any): value is object {
```

```
return typeof value === 'object' && value !== null;
}
const reactive = (obj) => {
// 传入非对象
if (!isPlainObj(obj)) {
return obj;
}
// 声明响应式数据
const proxy = new Proxy(obj, {
get(target, key, receiver) {
// 依赖收集方法 track
track(target, 'get', key);
// 调用 Reflect Api 获得原始的数据 你可以将它简单理解成为 target[key]
let result = Reflect.get(target, key, receiver);
// 依赖为对象 递归进行reactive处理
if (isPlainObj(result)) {
return reactive(result);
}
// 配合Reflect解决当访问get属性递归依赖this的问题
return result;
}
,
set() {
// dosomething
}
}
);
return proxy;
}
;
```

上边我们填充了在 Proxy 中的 get 陷阱的逻辑：

当访问响应式对象 proxy 中的属性时，首先会针对于对应的属性进行依赖收集。主要依靠的是 track 方法。

之后如果访问该响应式对象 key 对应的 value 仍为对象时，会再次递归调用 reactive 方法进行处理。

需要注意的是递归进行 reactive 时是一层懒处理，换句话说只有访问时才会递归处理并不是在初始化时就会针对于传入的 obj 进行递归处理。

当然这里的依赖收集主要依靠的就是 track 方法，我们会在稍后详解实现这个方法。

依赖收集

接下来我们来看看 set 陷阱中的逻辑，当触发对于 proxy 对象的属性修改时会触发 set 陷阱从而进行触发对应 Effect 的执行。

我们来一起看看对应的 set 陷阱中的逻辑：

```
// reactivity/src/reactivity.ts
export function isPlainObj(value: any): value is object {
return typeof value === 'object' && value !== null;
}
const reactive = (obj) => {
```

```
// 传入非对象
if (!isPlainObj(obj)) {
return obj;
}
// 声明响应式数据
const proxy = new Proxy(obj, {
get(target, key, receiver) {
// 依赖收集方法 track
track(target, 'get', key);
// 调用 Reflect Api 获得原始的数据 你可以将它简单理解成为 target[key]
let result = Reflect.get(target, key, receiver);
// 依赖为对象 递归进行reactive处理
if (isPlainObj(result)) {
return reactive(result);
}
// 配合Reflect解决当访问get属性递归依赖this的问题
return result;
}
,
// 当进行设置时进行触发更新
set(target, key, value, receiver) {
const oldValue = target[key];
// 配合Reflect解决当访问get属性递归依赖this的问题
const result = Reflect.set(target, key, value, receiver);
// 如果两次变化的值相同 那么不会触发更新
if (value !== oldValue) {
// 触发更新
trigger(target, 'set', key, value, oldValue);
}
return result;
}
,
}
);
return proxy;
}
;
```

同样，我们在上边填充了对应 set 陷阱之中的逻辑，当设置响应式对象时会触发对应的 set 陷阱。我们会在 set 陷阱中触发对应的 trigger 逻辑进行触发更新：将依赖的 effect 重新执行。

关于为什么我们在使用 Proxy 时需要配合 Refelct ，我在这篇文章有详细讲解。感兴趣的朋友可以查看这里 [为什么Proxy一定要配合Reflect使用？]。

上边我们完成了 reactive.ts 文件的基础逻辑，遗留了两个核心方法 track & trigger 方法。

在实现着两个方法之前，我们先来一起看看 effect 是如何被实现的。

effect 文件

effect 基础使用

让我们把视野切到 effcet.ts 中，我们稍微来回忆一下 effect Api 的用法：

```
const {
reactive,
effect
}
= Vue
const obj = {
name: '19Qingfeng'
}
// 创建响应式数据
const reactiveData = reactive(obj)
// 创建effect依赖响应式数据
effect(() => {
app.innerHTML = reactiveData.name
}
)
```

effect 基础原理

上边我们看到，effect Api 有以下特点：

effect 接受一个函数作为入参。

当调用effect(fn) 时，内部的函数会直接被调用一次。

其次，当 effect 中的依赖的响应式数据发生改变时。我们期望 effect 会重新执行，比如这里的 effect 依赖了 reactiveData.name 上的值。

接下来我们先来一起实现一个简单的 Effect Api：

```
function effect(fn) {
// 调用Effect创建一个的Effect实例
const _effect = new ReactiveEffect(fn);
// 调用Effect时Effect内部的函数会默认先执行一次
_effect.run();
// 创建effect函数的返回值：_effect.run() 方法（同时绑定方法中的this为_effect实例对象）
const runner = _effect.run.bind(_effect);
// 返回的runner函数挂载对应的_effect对象
runner.effect = _effect;
return runner;
}
```

这里我们创建了一个基础的 effect Api，可以看到它接受一个函数 fn 作为参数。

当我们运行 effect 时，会创建一个 const _effect = new ReactiveEffect(fn); 对象。

同时我们会调用 _effect.run() 这个实例方法立即执行传入的 fn ，之所以需要立即执行传入的 fn 我们在上边提到过：当代码执行到 effect(fn) 时，实际上会立即执行 fn 函数。

我们调用的 _effect.run() 实际内部也会执行 fn ，我们稍微回忆下上边的 Demo 当代码执行 effect(fn) 时候相当于执行了：

```
// ...
effect(() => {
app.innerHTML = reactiveData.name
}
)
```

会立即执行传入的 fn 也就是 () => {
app.innerHTML = reactiveData.name
}
会修改 app 节点中的内容。
同时，我们之前提到过因为 reactiveData 是一个 proxy 代理对象，当我们访问它的属性时实际上会触发它的 get 陷阱。

```
// effect.ts
export let activeEffect;
export function effect(fn) {
// 调用Effect创建一个的Effect实例
const _effect = new ReactiveEffect(fn);
// 调用Effect时Effect内部的函数会默认先执行一次
_effect.run();
// 创建effect函数的返回值：_effect.run() 方法（同时绑定方法中的this为_effect实例对象）
const runner = _effect.run.bind(_effect);
// 返回的runner函数挂载对应的_effect对象
runner.effect = _effect;
return runner;
}
/**
* Reactive Effect
*/
export class ReactiveEffect {
private fn: Function;
constructor(fn) {
this.fn = fn;
}
run() {
try {
activeEffect = this;
// run 方法很简单 就是执行传入的fn
return this.fn();
}
finally {
activeEffect = undefined
}
}
}
```

这是一个非常简单的 ReactiveEffect 实现，它的内部非常简单就是简单的记录了传入的 fn ，同时拥有一个 run 实例方法当调用 run 方法时会执行记录的 fn 函数。
同时，我们在模块内部声明了一个 activeEffect 的变量。当 我们调用运行 effect(fn) 时，实际上它会经历以下步骤：
首先用户代码中调用 effect(fn)
VueJs 内部会执行 effect 函数，同时创建一个 _effect 实例对象。立即调用 _effect.run() 实例方法。
重点就在所谓的 _effect.run() 方法中。

首先，当调用 _effect.run() 方法时，我们会执行 activeEffect = this 将声明的 activeEffect 变成当前对应的 _effect 实例对象。
同时，run() 方法接下来会调用传入的 fn() 函数。
当 fn() 执行时，如果传入的 fn() 函数存在 reactive() 包裹的响应式数据，那么实际上是会进入对应的 get 陷阱中。
当进入响应式数据的 get 陷阱中时，不要忘记我们声明全局的 activeEffect 变量，我们可以在对应响应式数据的 get 陷阱中拿到对应 activeEffect (也就是创建的 _effect) 变量。
接下来我们需要做的很简单:
在响应式数据的 get 陷阱中记录该数据依赖到的全局 activeEffect 对象(_effect)（依赖收集）也就是我们之前遗留的 track 方法。
同时:
当改变响应式数据时，我们仅仅需要找出当前对应的数据依赖的 _effect ，修改数据同时重新调用 _effect.run() 相当于重新执行了 effect（fn）中的 fn。那么此时不就是相当于修改数据页面自动更新吗？这一步就被称为依赖收集，也就是我们之前遗留的 trigger 方法。
track & trigger 方法
让我们会回到之前遗留的 track 和 trigger 逻辑中，接下来我们就尝试去实现它。
这里我们将在 effect.ts 中来实现这两个方法，将它导出提供给 reactive.ts 中使用。
思路梳理
上边我们提到过，核心思路是当代码执行到 effect(fn) 时内部会调用对应的 fn 函数执行。当 fn 执行时会触发 fn 中依赖的响应式数据的 get ，当 get 触发时我们记录到对应 声明的(activeEffect) _effect 对象和对应的响应式数据的关联即可。
当响应式数据改变时，我们取出关联的 _effect 对象，重新调用 _effect.run() 重新执行 effect(fn) 传入的 fn 函数即可。
看到这里，一些同学已经反应过来了。我们有一份记录对应 activeEffect(_effect) 和 对应的响应式数据的表，于是我们自然而然的想到使用一个 WeakMap 来存储这份关系。
之所以使用 WeakMap 来存储，第一个原因自然是我们需要存储的 key 值是非字符串类型这显然只有 map 可以。其次就是 WeakMap 的 key 并不会影响垃圾回收机制。
创建映射表
上边我们分析过，我们需要一份全局的映射表来维护 _effect 实例和依赖的响应式数据的关联:
于是我们自然想到通过一个 WeakMap 对象来维护映射关系，那么如何设计这个 WeakMap 对象呢？这里我就不卖关子了。
我们再来回忆下上述的 Demo ：

```
// ...
const {
reactive,
effect
}
= Vue
const obj = {
name: '19Qingfeng'
}
// 创建响应式数据
const reactiveData = reactive(obj)
// 创建effect依赖响应式数据
effect(() => {
app.innerHTML = reactiveData.name
```

```
}
)
// 上述Demo的基础上增加了一个effect依赖逻辑
effect(() => {
app2.innerHTML = reactiveData.name
}
)
```

首先针对于响应式数据 reactiveData 它是一个对象，上述代码中的 effect 中依赖了 reactiveData 对象上的 name 属性。

所以，我们仅仅需要关联当前响应式对象中的 name 属性和对应 effect 即可。

同时，针对于同一个响应式对象的属性比如这里的 name 属性被多个 effect 依赖。自然我们可以想到一份响应式数据的属性可以和多个 effect 依赖。

根据上述的分析最终 Vuejs 中针对于这份映射表设计出来了这样的结构：

当一个 effect 中依赖对应的响应式数据时，比如上述 Demo ：

我们创建的全局的 WeakMap 首先会将响应式对象的原始对象（未代理前的对象）作为 key ，value 为一个 Map 对象。

同时 effect 内部使用了上述对象的某个属性，那么此时 WeakMap 对象的该对象的值（我们刚才创建的 Map ）。我们会在这个 Map 对象中设置 key 为使用到的属性，value 为一个 Set 对象。

为什么对应属性的值为一个 Set ，这非常简单。因为该属性可能会被多个 effect 依赖到。所以它的值为一个 Set 对象，当该属性被某个 effect 依赖到时，会将对应 _effect 实例对象添加进入 Set 中。

也许有部分同学乍一看对于这份映射表仍然比较模糊，没关系接下来我会用代码来描述这一过程。你可以结合代码和这段文字进行一起理解。

track 实现

接下来我们来看看 track 方法的实现：

```
// *用于存储响应式数据和Effect的关系Hash表
const targetMap = new WeakMap();
/**
* 依赖收集函数 当触发响应式数据的Getter时会进入track函数
* @param target 访问的原对象
* @param type 表示本次track从哪里来
* @param key 访问响应式对象的key
*/
export function track(target, type, key) {
// 当前没有激活的全局Effect 响应式数据没有关联的effect 不用收集依赖
if (!activeEffect) {
return;
}
// 查找是否存在对象
let depsMap = targetMap.get(target);
if (!depsMap) {
targetMap.set(target, (depsMap = new Map()));
}
// 查找是否存在对应key对应的 Set effect
let deps = depsMap.get(key);
if (!deps) {
```

```
depsMap.set(key, (deps = new Set()));
}
// 其实Set本身可以去重 这里判断下会性能优化点
const shouldTrack = !deps.has(activeEffect) && activeEffect;
if (shouldTrack) {
// *收集依赖，将 effect 进入对应WeakMap中对应的target中对应的keys
deps.add(activeEffect);
}
}
```

我们一行一行分析上边的 track 方法，这个方法我们之前提到过。它是在 reactive.ts中对于某个响应式属性进行依赖收集（触发proxy的 get 陷阱）时触发的，忘记了的小伙伴可以翻回去重新看下。

首先，它会判断当前 activeEffect 是否存在，所谓 actvieEffect 也就是当前是否存在 effect 。换句话说，比如这样：

```
// ...
app.innerHTML = reactiveData.name
```

那么我们有必要进行依赖收集吗，虽然 reactiveData 是一个响应式数据这不假，但是我们并没有在模板上使用它。它并不存在任何关联的 effect ，所以完全没有必要进行依赖收集。

而在这种情况下:

```
effect(() => {
app.innerHTML = reactiveData.name
}
)
```

只有我们在 effect(fn) 中，当 fn 中使用到了对应的响应式数据。简单来说也就是 activeEffect 存在值得时候，对于响应式数据的依赖收集才有意义。

其次，接下来会去全局的 targetMap 中寻找是否已经存在对应响应式数据的原始对象 -> depsMap 。如果该对象首次被收集，那么我们需要在 targetMap 中设置 key 为 target ，value 为一个新的Map。

```
// 查找是否存在对象
let depsMap = targetMap.get(target);
if (!depsMap) {
// 不存在则创建一个Map作为value，将target作为key放入depsMap中
targetMap.set(target, (depsMap = new Map()));
}
```

同时我们会继续去上一步返回的 deps ，此时的 deps 是一个 Map 。它的内部会记录改对象中被进行依赖收集的属性。

我们回去寻找 name 属性是否存在，显然它是第一次进行依赖收集。所以会进行：

```
// 查找是否存在对应key对应的 Set effect
let deps = depsMap.get(key);
if (!deps) {
// 同样，不存在则创建set放入
depsMap.set(key, (deps = new Set()));
}
```

此时，比如上方的 Demo 中，当代码执行到 effect 中的 fn 碰到响应式数据的 get 陷阱时，触发 track 函数。

我们会为全局的 targetMap 对象中首先设置 key 为 obj （reactiveData的原始对象），

value 为一个 Map 。
其次，我们会为该创建的 Map 中再次进行设置 key 为该响应式对象需要被收集的属性，也就是我们在 effect 中访问到该对象的 name ，value 为一个 Set 集合。
接下里 Set 中存放什么其实很简单，我们仅需在对应 Set 中记录当前正在运行的 effct 实例对象，也就是 activeEffct 就可以达到对应的依赖收集效果。
此时，targetMap 中就会存放对应的对象和关联的 effect 了。
trigger 实现
当然，上述我们已经通过对应的 track 方法收集了相关响应式数据和对应它依赖的 effect 。
那么接下来如果当改变响应式数据时（触发 set 陷阱时）自然我们仅仅需要找到对应记录的 effect 对象，调用它的 effect.run() 重新执行不就可以让页面跟随数据改变而改变了吗。
我们来一起看看 trigger 方法：

```
// ... effect.ts
/**
* 触发更新函数
* @param target 触发更新的源对象
* @param type 类型
* @param key 触发更新的源对象key
* @param value 触发更新的源对象key改变的value
* @param oldValue 触发更新的源对象原始的value
*/
export function trigger(target, type, key, value, oldValue) {
// 简单来说 每次触发的时 我拿出对应的Effect去执行 就会触发页面更新
const depsMap = targetMap.get(target);
if (!depsMap) {
return;
}
let effects = depsMap.get(key);
if (!effects) {
return;
}
effects = new Set(effects);
effects.forEach((effect) => {
// 当前zheng
if (activeEffect !== effect) {
// 默认数据变化重新调用effect.run()重新执行清空当前Effect依赖重新执行Effect中fn进行重新收集依赖以及视图更新
effect.run();
}
}
);
}
```

接下来我们在 effect.ts 中来补充对应的 trigger 逻辑，其实 trigger 的逻辑非常简单。每当响应式数据触发 set 陷阱进行修改时，会触发对应的 trigger 函数。
他会接受对应的 5 个 参数，我们在函数的注释中已经标明了对应的参数。
当触发响应式数据的修改时，首先我们回去 targetMap 中寻找 key 为对应原对象的值，自然因为在 track 中我们已经保存了对应的值，所以当然可以拿到一个 Map 对象。

因为该 Map 对象中存在对应 key 为 name 值为该属性依赖的 effect 的 Set 集合，所以我们仅需要依次拿出对应修改的属性，比如我们调用：

```
// ...
const {
reactive,
effect
}
= Vue
const obj = {
name: '19Qingfeng'
}
// 创建响应式数据
const reactiveData = reactive(obj)
// 创建effect依赖响应式数据
effect(() => {
app.innerHTML = reactiveData.name
}
)
// 修改响应式数据 触发set陷阱
reactiveData.name = 'wang.haoyu'
```

当我们调用 reactiveData.name = 'wang.haoyu' 时，我们会一层一层取到 targetMap 中 key 为 obj 的 depsMap（Map） 对象。

再从 depsMap 中拿到 key 为 name 属性的 Set 集合（Set 中保存该响应式对象属性依赖的 effect）。

迭代当前 Set 中的所有 effect 进行 effect.run() 重新执行 effect 对象中记录的 fn 函数。

因为我们在 reactive.ts 中的 set 陷阱中对于数据已经修改之后调用了 trigger 方法，trigger 导致重新执行 effect(fn) 中的 fn，所以自然而然 fn() 重新执行 app.innerHTML 就会变成最新的 wang.haoyu 。

整个响应式核心原理其实一点都不难对吧，核心思想还是文章开头的那句话：对于数据访问时，需要进行依赖收集。记录当前数据中依赖了哪些 Effect ，当进行数据修改时候同样会进行触发更新，重新执行当前数据依赖的 Effect。

阶段总结

其实写到这里已经 8K 多字了，原本打算是和大家过一遍整个 Vue 3.2 中关于 reactivity 的逻辑，包括各种边界情况。

比如文章中的代码其实仅仅只能说是实现了一个乞丐版的响应式原理，其他一些边界情况，比如：

多个 effect 嵌套时的处理。

多次 reactive 调用同一对象，或者对于已经 reactive 包裹的响应式对象。

每次触发更新时，对于前一次依赖收集的清理。

shallow、readonly 情况等等...

这些边界情况其实文章中的代码我都没有考虑，如果后续有小伙伴对这方面感兴趣我会再次开一篇文章去继续这次的代码去实现一个完整的 reactive 方法。

不过，透过现象看本质。VueJs 中所谓主打的数据响应式核心原理即是文章中代码所表现的思想。

我在这个代码地址，也自己实现了一版比较完整的精简版 reactivity 模块，有兴趣的同学可以自行查阅。

当然，你也可以直接参照源代码进行阅读。毕竟 Vue 3 的代码相较于 2 来说其实已经很人性化了。

面试题 7. 简述nextTick 的作用是什么？他的实现原理是什么？

推荐指数：★★★★ 试题难度：初级 试题类型：原理题

试题回答参考思路：

作用 ：vue 更新 DOM 是异步更新的，数据变化，DOM 的更新不会马上完成， nextTick 的回调是在下次 DOM 更新循环结束之后执行的延迟回调 。
实现原理 ：nextTick 主要使用了 宏任务和微任务 。根据执行环境分别尝试采用
Promise：可以将函数延迟到当前函数调用栈最末端
MutationObserver ： 是 H5 新加的一个功能，其功能是监听 DOM 节点的变动，在所有 DOM 变动完成后，执行回调函数setImmediate：用于中断长时间运行的操作，并在浏览器完成其他操作（如事件和显 示更新）后立即运行回调函数
如果以上都不行则采用 setTimeout 把函数延迟到 DOM 更新之后再使用，原因是宏任务消耗大于微任务，优先使用微任务，最后使用消耗最大的宏任务。

面试题 8. 请简述Vue 的性能优化可以从哪几个方面去思考设计？

推荐指数：★★★★★ 试题难度：高难 试题类型：原理题

试题回答参考思路：

这里只列举针对 Vue 的性能优化，整个项目的性能优化是一个大工程。
对象层级不要过深，否则性能就会差。
不需要响应式的数据不要放在 data 中（可以使用 Object.freeze() 冻结数据）
v-if 和 v-show 区分使用场景
computed 和 watch 区分场景使用
v-for 遍历必须加 key，key最好是id值，且避免同时使用 v-if
大数据列表和表格性能优化 - 虚拟列表 / 虚拟表格
防止内部泄露，组件销毁后把全局变量和时间销毁
图片懒加载
路由懒加载
异步路由
第三方插件的按需加载
适当采用 keep-alive 缓存组件
防抖、节流的运用
服务端渲染 SSR or 预渲染

面试题 9. 请叙述Vue 中使用了哪些设计模式?

推荐指数：★★★★ 试题难度：中级 试题类型：原理题

试题回答参考思路：

1 工厂模式 - 传入参数即可创建实例 虚拟 DOM 根据参数的不同返回基础标签的 Vnode 和组件 Vnode。
2 单例模式 - 整个程序有且仅有一个实例 vuex 和 vue-router 的插件3注册方法 install 判断如果系统存在实例就直接返回掉。
3 发布-订阅模式。（vue 事件机制）
4 观察者模式。（响应式数据原理）
5 装饰器模式（@装饰器的用法）
6 策略模式，策略模式指对象有某个行为，但是在不同的场景中，该行为有不同的实现方案 - 比如选项的合并策略。

面试题 10. Vuex 页面刷新数据丢失怎么解决?

推荐指数: ★★★★ 试题难度: 初级 试题类型: 原理题

试题回答参考思路:

需要做 vuex 数据持久化 ，一般使 用本地储存的方案 来保存数据，可以自己设计存储方案，也可以使用第三方插件。
推荐使用 vuex-persist（ 脯肉赛斯特 ）插件，它是为 Vuex 持久化储存而生的一个插件。不需要你手动存取 storage ， 而是直接将状态保存至 cookie 或者 localStorage 中

面试题 11. 简述如何使用Vue-router实现懒加载的方式?

推荐指数: ★★★★★ 试题难度: 中级 试题类型: 原理题

试题回答参考思路:

vue异步组件
vue异步组件技术 ==== 异步加载
vue-router配置路由 ，使用vue的异步组件技术 ，可以实现按需加载 。但是,这种情况下一个组件生成一个js文件

```
/* vue异步组件技术 */
{
path: '/home',
name: 'home',
component: resolve => require(['@/components/home'],resolve)
},{
path: '/index',
name: 'Index',
component: resolve => require(['@/components/index'],resolve)
},{
path: '/about',
name: 'about',
component: resolve => require(['@/components/about'],resolve)
}
```

es提案的import()

路由懒加载(使用import)

```
// 下面2行代码，没有指定webpackChunkName，每个组件打包成一个js文件。
/* const Home = () => import('@/components/home')
const Index = () => import('@/components/index')
const About = () => import('@/components/about') */
// 下面2行代码，指定了相同的webpackChunkName，会合并打包成一个js文件。把组件按组分块
const Home = () => import(/* webpackChunkName: 'ImportFuncDemo' */ '@/components/home')
const Index = () => import(/* webpackChunkName: 'ImportFuncDemo' */ '@/components/index')
const About = () => import(/* webpackChunkName: 'ImportFuncDemo' */ '@/components/about')
```

```
{
path: '/about',
component: About
}, {
path: '/index',
component: Index
}, {
path: '/home',
component: Home
}
```

webpack的require,ensure()

vue-router配置路由，使用webpack的require.ensure技术，也可以实现按需加载。这种情况下，多个路由指定相同的chunkName，会合并打包成一个js文件。

```
/* 组件懒加载方案三: webpack提供的require.ensure() */
{
path: '/home',
name: 'home',
component: r => require.ensure([], () => r(require('@/components/home')), 'demo')
}, {
path: '/index',
name: 'Index',
component: r => require.ensure([], () => r(require('@/components/index')), 'demo')
}, {
path: '/about',
name: 'about',
component: r => require.ensure([], () => r(require('@/components/about')), 'demo-01')
}
```

面试题 12. Vue首屏白屏如何解决?

推荐指数: ★★ 试题难度: 高难 试题类型: 原理题

试题回答参考思路:

1）路由懒加载
2）vue-cli开启打包压缩 和后台配合 gzip访问
3）进行cdn加速
4）开启vue服务渲染模式
5）用webpack的externals属性把不需要打包的库文件分离出去，减少打包后文件的大小
6）在生产环境中删除掉不必要的console.log

```
plugins: [
new webpack.optimize.UglifyJsPlugin({ //添加-删除console.log
compress: {
warnings: false,
drop_debugger: true,
drop_console: true
},
sourceMap: true
})
```

7）开启nginx的gzip ,在nginx.conf配置文件中配置

```
http { //在 http中配置如下代码,
gzip on;
gzip_disable "msie6";
gzip_vary on;
gzip_proxied any;
gzip_comp_level 8; #压缩级别
gzip_buffers 16 8k;
#gzip_http_version 1.1;
gzip_min_length 100; #不压缩临界值
gzip_types text/plain application/javascript application/x-javascript text/css
application/xml text/javascript application/x-httpd-php image/jpeg image/gif
image/png;
}
```

8）添加loading效果，给用户一种进度感受

面试题 13. Vie3.0 Proxy 相比 defineProperty 的优势在哪里?

推荐指数: ★★★★ 试题难度: 高难 试题类型: 原理题

试题回答参考思路:

Vue3.x 改用 Proxy 替代 Object.defineProperty
原因在于 Object.defineProperty 本身存在的一 些问题 ：
Object.defineProperty 只能劫持对象属性的 getter 和 setter 方法。
Object.definedProperty 不支持数组(可以监听数组,不过数组方法无法监听自己重写)，更准确的说是不支持数组的各种 API(所以 Vue 重写了数组方法。
而相比 Object.defineProperty，Proxy 的优点在于：
Proxy 是直接代理劫持整个对象。
Proxy 可以直接监听对象和数组的变化，并且有多达 13 种拦截方法。
目前，Object.definedProperty 唯一比 Proxy 好的一点就是兼容性，不过 Proxy 新标准也受到浏览器厂商重点持续的性能优化当中

面试题 14. vue通过数据劫持可以精准的探测数据变化，为什么还要进行diff检测差异?

推荐指数：★★★★★ 试题难度： 中级 试题类型： 原理题

试题回答参考思路：

现代前端框架有两种方式侦测变化,一种是pull一种是push

pull: 其代表为React,我们可以回忆一下React是如何侦测到变化的,我们通常会用setStateAPI显式更新,然后React会进行一层层的Virtual Dom Diff操作找出差异,然后Patch到DOM上,React从一开始就不知道到底是哪发生了变化,只是知道「有变化了」,然后再进行比较暴力的Diff操作查找「哪发生变化了」，另外一个代表就是Angular的脏检查操作。

push: Vue的响应式系统则是push的代表,当Vue程序初始化的时候就会对数据data进行依赖的收集,一但数据发生变化,响应式系统就会立刻得知,因此Vue是一开始就知道是「在哪发生变化了」,但是这又会产生一个问题,如果你熟悉Vue的响应式系统就知道,通常一个绑定一个数据就需要一个Watcher,一但我们的绑定细粒度过高就会产生大量的Watcher,这会带来内存以及依赖追踪的开销,而细粒度过低会无法精准侦测变化,因此Vue的设计是选择中等细粒度的方案,在组件级别进行push侦测的方式,也就是那套响应式系统,通常我们会第一时间侦测到发生变化的组件,然后在组件内部进行Virtual Dom Diff获取更加具体的差异,而Virtual Dom Diff则是pull操作,Vue是push+pull结合的方式进行变化侦测的.

面试题 15. 简述Vue的普通Slot以及作用域Slot的区别 ?

推荐指数：★★★★★ 试题难度： 中级 试题类型： 原理题

试题回答参考思路：

普通插槽
普通插槽是渲染后做替换的工作。父组件渲染完毕后，替换子组件的内容。

作用域插槽
作用域插槽可以拿到子组件里面的属性。在子组件中传入属性然后渲染。

// 有name的属于具名插槽，没有name属于匿名插槽

xxxx

xxxx

普通插槽渲染的位置是在它的父组件里面，而不是在子组件里面
作用域插槽渲染是在子组件里面
1.插槽slot

在渲染父组件的时候，会将插槽中的先渲染。

创建组件虚拟节点时，会将组件的儿子的虚拟节点保存起来。当初始化组件时,通过插槽属性将儿 子进行分类 {a:[vnode],b[vnode]}渲染组件时会拿对应的slot属性的节点进行替换操作。（插槽的作用域为父组件，插槽中HTML模板显示不显示、以及怎样显示由父组件来决定）

有name的父组件通过html模板上的slot属性关联具名插槽。没有slot属性的html模板默认关联匿名插槽。

2.作用域插槽slot-scope

作用域插槽在解析的时候，不会作为组件的孩子节点。会解析成函数，当子组件渲染时，会调用此函数进行渲染。

或者可以说成作用域插槽是子组件可以在slot标签上绑定属性值，在父组件可以拿到子组件的数据，通过子组件绑定数据传递给父组件。（插槽的作用域为子组件）

子组件：

父组件：

面试题 16. Vue路由的使用以及如何解决在router-link上添加点击事件无效的情况？

推荐指数：★★★★ 试题难度：高难 试题类型：原理题

试题回答参考思路：

使用@click.native。原因:router-link会阻止click事件,.native指直接监听一个原生事件

面试题 17. 请说明Vue的solt的用法?

推荐指数: ★★★★★ 试题难度: 高难 试题类型: 原理题

试题回答参考思路:

在子组件内使用特殊的＜slot>元素就可以为这个子组件开启一个slot（插槽），在父组件模板里，插入在子组件标签内的所有内容将替代子组件的＜slot> 标签及它的内容。

简单说来就是：在子组件内部用 标签占位，当在父组件中使用子组件的时候，我们可以在子组件中插入内容，而这些插入的内容则会替换 标签的位置。

当然：单个solt的时候可以不对solt进行命名，如果存在多个 则一个可以不命名，其他必须命名，在调用的时候指定名称的对应替换slot，没有指定的则直接默认无名称的solt

面试题 18. 简述Vuex和单纯的全局对象有什么区别?

推荐指数: ★★★★★ 试题难度: 高难 试题类型: 原理题

试题回答参考思路:

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。
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。

面试题 19. 简述为什么Vue采用异步渲染？

推荐指数: ★★★★★ 试题难度: 中级 试题类型: 原理题

试题回答参考思路：

vue是组件级更新，当前组件里的数据变了，它就会去更新这个组件。当数据更改一次组件就要重新渲染一次，性能不高，为了防止数据一更新就更新组件，所以做了个异步更新渲染。（核心的方法就是nextTick）

源码实现原理：
当数据变化后会调用notify方法，将watcher遍历，调用update方法通知watcher进行更新，这时候watcher并不会立即去执行，在update中会调用queueWatcher方法将watcher放到了一个队列里，在queueWatcher会根据watcher的进行去重，多个属性依赖一个watcher，如果队列中没有该watcher就会将该watcher添加到队列中，然后通过nextTick异步执行flushSchedulerQueue方法刷新watcher队列。flushSchedulerQueue中开始会触发一个before的方法，其实就是beforeUpdate，然后watcher.run() 才开始真正执行watcher，执行完页面就渲染完成啦，更新完成后会调用updated钩子。

面试题 20. Vue中data的属性可以和methods中方法同名吗，为什么?

推荐指数：★★★ 试题难度：中级 试题类型：原理题

试题回答参考思路：

可以同名，methods的方法名会被data的属性覆盖；调试台也会出现报错信息，但是不影响执行；
原因：源码定义的initState函数内部执行的顺序：props>methods>data>computed>watch

```
//initState部分源码
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)
}
}
```

面试题 21. 简述Proxy 与 Object.defineProperty 优劣对比?

推荐指数：★★★★ 试题难度：高难 试题类型：原理题

试题回答参考思路：

1.Proxy 可以直接监听对象而非属性；
2.Proxy 可以直接监听数组的变化；
3.Proxy 有多达 13 种拦截方法,不限于 apply、ownKeys、deleteProperty、has 等等是 Object.defineProperty 不具备的；
4.Proxy 返回的是一个新对象，我们可以只操作新的对象达到目的，而 Object.defineProperty 只能遍历对象属性直接修改；
5.Proxy 作为新标准将受到浏览器厂商重点持续的性能优化，也就是传说中的新标准的性能红利；
6.Object.defineProperty 的优势如下:
兼容性好，支持 IE9，而 Proxy 的存在浏览器兼容性问题,而且无法用 polyfill 磨平，因此 Vue 的作者才声明需要等到下个大版本(3.0)才能用 Proxy 重写。

面试题 22. 简述Vue初始化过程中（new Vue(options)）都做了什么?

推荐指数： ★★★★ 试题难度： 高难 试题类型： 原理题

试题回答参考思路：

处理组件配置项；初始化根组件时进行了选项合并操作，将全局配置合并到根组件的局部配置上；初始化每个子组件时做了一些性能优化，将组件配置对象上的一些深层次属性放到 vm.$options 选项中，以提高代码的执行效率；
初始化组件实例的关系属性，比如 $parent、parent、parent、children、$root、root、root、refs 等
处理自定义事件
调用 beforeCreate 钩子函数
初始化组件的 inject 配置项，得到 ret[key] = val 形式的配置对象，然后对该配置对象进行响应式处理，并代理每个 key 到 vm 实例上
数据响应式，处理 props、methods、data、computed、watch 等选项
解析组件配置项上的 provide 对象，将其挂载到 vm._provided 属性上
调用 created 钩子函数
如果发现配置项上有 el 选项，则自动调用 $mount 方法，也就是说有了 el 选项，就不需要再手动调用 $mount 方法，反之，没提供 el 选项则必须调用 $mount
接下来则进入挂载阶段

```
// core/instance/init.js
export function initMixin (Vue: Class) {
Vue.prototype._init = function (options?: Object) {
const vm: Component = this
vm._uid = uid++

// 如果是Vue的实例，则不需要被observe
vm._isVue = true

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 (process.env.NODE_ENV !== 'production') {
initProxy(vm)
} else {
vm._renderProxy = vm
}

vm._self = vm

initLifecycle(vm)
initEvents(vm)
callHook(vm, 'beforeCreate')
initInjections(vm) // resolve injections before data/props

initState(vm)
initProvide(vm) // resolve provide after data/props
callHook(vm, 'created')

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

面试题 23. SPA首屏加载速度慢的怎么解决?

推荐指数: ★★★ 试题难度: 中级 试题类型: 原理题

试题回答参考思路:

请求优化：CDN 将第三方的类库放到 CDN 上，能够大幅度减少生产环境中的项目体积，另外 CDN 能够实时地根据网络流量和各节点的连接、负载状况以及到用户的距离和响应时间等综合信息将用户的请求重新导向离用户最近的服务节点上。

缓存：将长时间不会改变的第三方类库或者静态资源设置为强缓存，将 max-age 设置为一个非常长的时间，再将访问路径加上哈希达到哈希值变了以后保证获取到最新资源，好的缓存策略有助于减轻服务器的压力，并且显著的提升用户的体验

gzip：开启 gzip 压缩，通常开启 gzip 压缩能够有效的缩小传输资源的大小。

http2：如果系统首屏同一时间需要加载的静态资源非常多，但是浏览器对同域名的 tcp 连接数量是有限制的(chrome 为 6 个)超过规定数量的 tcp 连接，则必须要等到之前的请求收到响应后才能继续发送，而 http2 则可以在多个 tcp 连接中并发多个请求没有限制，在一些网络较差的环境开启 http2 性能提升尤为明显。

懒加载：当 url 匹配到相应的路径时，通过 import 动态加载页面组件，这样首屏的代码量会大幅减少，webpack 会把动态加载的页面组件分离成单独的一个 chunk.js 文件

预渲染：由于浏览器在渲染出页面之前，需要先加载和解析相应的 html、css 和 js 文件，为此会有一段白屏的时间，可以添加loading，或者骨架屏幕尽可能的减少白屏对用户的影响体积优化
合理使用第三方库：对于一些第三方 ui 框架、类库，尽量使用按需加载，减少打包体积

使用可视化工具分析打包后的模块体积：webpack-bundle- analyzer 这个插件在每次打包后能够更加直观的分析打包后模块的体积，再对其中比较大的模块进行优化

提高代码使用率：利用代码分割，将脚本中无需立即调用的代码在代码构建时转变为异步加载的过程

封装：构建良好的项目架构，按照项目需求就行全局组件，插件，过滤器，指令，utils 等做一 些公共封装，可以有效减少我们的代码量，而且更容易维护资源优化

图片懒加载：使用图片懒加载可以优化同一时间减少 http 请求开销，避免显示图片导致的画面抖动，提高用户体验使用 svg 图标：相对于用一张图片来表示图标，svg 拥有更好的图片质量，体积更小，并且不需要开启额外的 http 请求

压缩图片：可以使用 image-webpack-loader，在用户肉眼分辨不清的情况下一定程度上压缩图片

面试题 24. 简述MINI UI是什么？怎么使用？说出至少三个组件的使用方法？

推荐指数：★★★ 试题难度：中级 试题类型：原理题

试题回答参考思路：

//基于vue框架的ui组件库很多，这里主要简单阐述一下组件的使用方法。
基于vue前端的组件库。使用npm安装，然后import样式和js；
vue.use(miniUi)全局引入。在单个组件局部引入；
import {Toast} form ‘mini-ui’;
组件一: Toast（‘登录成功’）
组件二： mint-header;
组件三：mint-swiper

面试题 25. 如何解决Vuex页面刷新数据丢失？

推荐指数：★★★★★ 试题难度：高难 试题类型：原理题

试题回答参考思路：

F5页面刷新，页面销毁之前的资源，重新请求，因此写在生命周期里的vuex数据是重新初始化，无法获取的，这也就是为什么会打印出空的原因。
解决思路1：
使用Localstorage sessionStorage 或cookie
实际使用时当vuex值变化时，F5刷新页面，vuex数据重置为初始状态，所以还是要用到localStorage，
解决方法2：
插件vuex-persistedstate
vuex-persistedstate默认持久化所有state，可以指定需要持久化的state

面试题 26. 简述如何在vue-cli生产环境使用全局常量？

推荐指数：★★ 试题难度：初级 试题类型：原理题

试题回答参考思路：

第一步，在 static 下新建 config.js：
第二步，在 config.js 里面设置全局变量：
第三步，在 index.html 里面引入：
第四步，在其他 .js 文件中即可使用：
第五步，打包后修改：通过 `npm run build` 命令打包后，此 config.js 文件会被打包到 `dist/static`文件夹下，此时如果需要修改 `PUBLIC_IP`，打开`config.js`即可修改，无需重新打包

面试题 27. 简述vue如何监听键盘事件中的按键?

推荐指数：★★★ 试题难度：初级 试题类型：原理题

试题回答参考思路：

在我们的项目经常需要监听一些键盘事件来触发程序的执行，而Vue中允许在监听的时候添加关键修饰符：
对于一些常用键，还提供了按键别名：
全部的按键别名：
.enter
.tab
.delete（捕获“删除”和“退格”键）
.esc
.space
.up
.down
.left
.right

修饰键：
.ctrl
.alt
.shift
.meta
与按键别名不同的是，修饰键和 keyup 事件一起用时，事件引发时必须按下正常的按键。换一种说法：如果要引发 keyup.ctrl，必须按下 ctrl 时释放其他的按键；单单释放 ctrl 不会引发事件

对于elementUI的input，我们需要在后面加上.native，因为elementUI对input进行了封装，原生的事件不起作用。

“昵称”

面试题 28. 请问什么是SSR ，它主要解决什么问题?

推荐指数： ★★★ 试题难度： 中级 试题类型： 原理题

试题回答参考思路：

Server-Side Rendering 我们称其为SSR，意为服务端渲染指由服务侧完成页面的 HTML 结构拼接的页面处理技术，发送到浏览器，然后为其绑定状态与事件，成为完全可交互页面的过程；

解决了以下两个问题：

seo：搜索引擎优先爬取页面HTML结构，使用ssr时，服务端已经生成了和业务想关联的HTML，有利于seo
首屏呈现渲染：用户无需等待页面所有js加载完成就可以看到页面视图（压力来到了服务器，所以需要权衡哪些用服务端渲染，哪些交给客户端）
缺点

复杂度：整个项目的复杂度
性能会受到影响
服务器负载变大，相对于前后端分离务器只需要提供静态资源来说，服务器负载更大，所以要慎重使用

面试题 29. Vue 3.0 所采用的 Composition Api 与 Vue 2.x使用的Options Api 有什么区别?

推荐指数： ★★★★★ 试题难度： 高难 试题类型： 原理题

试题回答参考思路：

Options Api

包含一个描述组件选项（data、methods、props等）的对象 options；

API开发复杂组件，同一个功能逻辑的代码被拆分到不同选项 ；
使用mixin重用公用代码，也有问题：命名冲突，数据来源不清晰；

Composition Api

vue3 新增的一组 api，它是基于函数的 api，可以更灵活的组织组件的逻辑。
解决options api在大型项目中，options api不好拆分和重用的问题

面试题 30. 请解释axios 是什么，其特点和常用语法 ？

推荐指数： ★★★ 试题难度： 中级 试题类型： 原理题

试题回答参考思路：

axios 是什么?

Axios 是一个基于 promise 的 HTTP 库，可以用在浏览器和 node.js 中。前端最流行的 ajax 请求库，
react/vue 官方都推荐使用 axios 发 ajax 请求
特点:

基于 promise 的异步 ajax 请求库，支持promise所有的API
浏览器端/node 端都可以使用，浏览器中创建XMLHttpRequests
支持请求 / 响应拦截器
支持请求取消
可以转换请求数据和响应数据，并对响应回来的内容自动转换成 JSON类型的数据
批量发送多个请求
安全性更高，客户端支持防御 XSRF，就是让你的每个请求都带一个从cookie中拿到的key，根据浏览器同源策略，假冒的网站是拿不到你cookie中得key的，这样，后台就可以轻松辨别出这个请求是否是用户在假冒网站上的误导输入，从而采取正确的策略。

常用语法:
axios(config): 通用/最本质的发任意类型请求的方式
axios(url[, config]): 可以只指定 url 发 get 请求
axios.request(config): 等同于 axios(config)
axios.get(url[, config]): 发 get 请求
axios.delete(url[, config]): 发 delete 请求
axios.post(url[, data, config]): 发 post 请求
axios.put(url[, data, config]): 发 put 请求
axios.defaults.xxx: 请求的默认全局配置
axios.interceptors.request.use(): 添加请求拦截器
axios.interceptors.response.use(): 添加响应拦截器
axios.create([config]): 创建一个新的 axios(它没有下面的功能)
axios.Cancel(): 用于创建取消请求的错误对象
axios.CancelToken(): 用于创建取消请求的 token 对象
axios.isCancel(): 是否是一个取消请求的错误

axios.all(promises): 用于批量执行多个异步请求
axios.spread(): 用来指定接收所有成功数据的回调函数的方法

面试题 31. 为什么 Vuex 的 mutation 中不能做异步操作?

推荐指数: ★★★★ 试题难度: 高难 试题类型: 原理题

试题回答参考思路:

每个mutation执行完成后都会对应到一个新的状态变更，这样devtools就可以打个快照存下来，然后就可以实现 time-travel 了。如果mutation支持异步操作，就没有办法知道状态是何时更新的，无法很好的进行状态的追踪，给调试带来困难

Mutation 必须是同步函数一条重要的原则就是要记住 mutation 必须是同步函数。为什么？请参考下面的例子：
mutations: { someMutation (state) { api.callAsyncMethod(() =>
{ state.count++ }) }}
现在想象，我们正在 debug 一个 app 并且观察 devtool 中的 mutation 日志。
每一条 mutation 被记录，devtools 都需要捕捉到前一状态和后一状态的快照。
然而，在上面的例子中 mutation 中的异步函数中的回调让这不可能完成：因为当 mutation 触发的时候，回调函数还没有被调用，devtools 不知道什么时候回调函数实际上被调用——实质上任何在回调函数中进行的状态的改变都是不可追踪的。
在组件中提交 Mutation 你可以在组件中使用 this.$store.commit('xxx') 提交 mutation，或者使用 mapMutations 辅助函数将组件中的 methods 映射为 store.commit 调用（需要在根节点注入 store）。
import { mapMutations } from 'vuex'export default {
// ... methods: {
...mapMutations([
'increment', // 将 `this.increment()` 映射为
`this.$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')`
})
}
}

面试题 32. 简述Vue 的 computed 的实现原理 ?

推荐指数: ★★★★★ 试题难度: 中级 试题类型: 原理题

试题回答参考思路:

1、每个computed属性都会生成对应的Watcher实例，watcher拥有value属性和get方法，computed的getter函数会在get方法中调用，并返回赋值给value。初始设置dirty和lazy为true，当lazy为true时不会立即执行get方法，而是会在读取computed值时执行；

2、将computed属性添加到组件实例上，通过get、set进行属性值的获取或设置，并且重新定义getter方法；

3、页面初始化时，会读取computed属性值，触发重新定义的getter，由于观察者的dirty值为true，将会调用原始的getter函数，当getter方法读取data数据时会触发原始的get方法(数据劫持中的get方法)，将computed对应的watcher添加到data依赖收集器(dep)中。观察者的get方法执行完后，更新观察者的value，并将dirty置为false，表示value值已更新，之后执行观察者的depend方法，将上层观察者也添加到getter函数中data的依赖收集器(dep)中，最后返回computed的value值；

4、当更改了computed属性getter函数依赖的data值时，将会触发之前dep收集的watcher，依次调用watcher的update方法，先调用computed的观察者的update方法，由于lazy为true，会将dirty先设置为true，表示computed属性getter函数依赖data发生变化，但不调用观察者的get方法更新value值。这时调用包含更新页面方法的观察者的update方法，在更新页面时会读取computed属性值，触发重新定义的getter函数，由于dirty为true，调用该观察者的get方法，更新value并返回，完成页面渲染；

面试题 33. 简述Vue SSR 的实现原理？

推荐指数：★★★★ 试题难度：中级 试题类型：原理题

试题回答参考思路：

app.js 作为客户端与服务端的公用入口，导出 Vue 根实例，供客户端 entry 与服务端 entry 使用。客户端 entry 主要作用挂载到 DOM 上，服务端 entry 除了创建和返回实例，还需要进行路由匹配与数据预获取。
webpack 为客服端打包一个 ClientBundle，为服务端打包一个 ServerBundle。
服务器接收请求时，会根据 url，加载相应组件，获取和解析异步数据，创建一个读取 Server Bundle 的 BundleRenderer，然后生成 html 发送给客户端。
客户端混合，客户端收到从服务端传来的 DOM 与自己的生成的 DOM 进行对比，把不相同的 DOM 激活，使其可以能够响应后续变化，这个过程称为客户端激活（也就是转换为单页应用）。为确保混合成功，客户 端与服务器端需要共享同一套数据。在服务端，可以在渲染之前获取数据，填充到 store 里，这样，在客户端挂载到 DOM 之前，可以直接从 store 里取数据。首屏的动态数据通过 window._INITIAL_STATE_ 发送到客户端
VueSSR 的原理，主要就是通过 vue-server-renderer 把 Vue 的组件输出成一个完整 HTML，输出到客户端，到达客户端后重新展开为一个单页应用。

面试题 34. 请简述构建 vue-cli 工程都用到了哪些技术？他们的作用分别是什么？

推荐指数：★★ 试题难度：中级 试题类型：原理题

试题回答参考思路：

vue.js：vue-cli 工程的核心，主要特点是双向数据绑定和组件系统。
vue-router：vue 官方推荐使用的路由框架。
vuex：专为 Vue.js 应用项目开发的状态管理器，主要用于维护 vue 组件间共用的一些 变量 和 方法。
axios（或者 fetch、ajax）：用于发起 GET 、或 POST 等 http请求，基于 Promise 设计。
vux等：一个专为vue设计的移动端UI组件库。
webpack：模块加载和vue-cli工程打包器。
eslint：代码规范工具
vue-cli 工程常用的 npm 命令有哪些?
下载 node_modules 资源包的命令：npm install
启动 vue-cli 开发环境的 npm命令：npm run dev
vue-cli 生成 生产环境部署资源 的 npm命令：npm run build
用于查看 vue-cli 生产环境部署资源文件大小的 npm命令：npm run build --report

面试题 35. Vue 中如何进行组件的使用？Vue 如何实现全局组件的注册?

推荐指数：★★★★ 试题难度：中级 试题类型：原理题

试题回答参考思路：

要使用组件，首先需要使用 import 来引入组件，然后在 components 属性中注册组件，之后就可以在模板中使用组件了。
可以使用 Vue.component 方法来实现全局组件的注册。

面试题 36. 简述vue2.x 和 vuex3.x 渲染器的 diff 算法 ?

推荐指数：★★★★★ 试题难度：高难 试题类型：原理题

试题回答参考思路：

简单来说，diff 算法有以下过程

同级比较，再比较子节点
先判断一方有子节点一方没有子节点的情况(如果新的 children 没有子节点，将旧的子节点移除)
比较都有子节点的情况(核心 diff)
递归比较子节点
正常 Diff 两个树的时间复杂度是 O(n^3)，但实际情况下我们很少会进行跨层级的移动 DOM，所以 Vue 将 Diff 进行了优化，从O(n^3) -> O(n)，只有当新旧 children 都为多个子节点时才需要用核心的 Diff 算法进行同层级比较。

Vue2 的核心 Diff 算法采用了双端比较的算法，同时从新旧 children 的两端开始进行比较，借助 key 值找到可复用的节点，再进行相关操作。相比 React 的 Diff 算法，同样情况下可以减少移动节点次数，减少不必要的性能损耗，更加的优雅。

Vue3.x 借鉴了 ivi 算法和 inferno 算法

在创建 VNode 时就确定其类型，以及在 mount/patch 的过程中采用位运算来判断一个 VNode 的类型，在这个基础之上再配合核心的 Diff 算法，使得性能上较 Vue2.x 有了提升。该算法中还运用了动态规划的思想求解最长递归子序列。

面试题 37. 怎么定义vue-router的动态路由以及如何获取传过来的动态参数?

推荐指数: ★★★★★ 试题难度: 中级 试题类型: 原理题

试题回答参考思路:

在router目录下的index.js文件中，对path属性加上/：id
使用router对象的params id
Vue中,如何用watch去监听router变化
当路由发生变化的时候，在watch中写具体的业务逻辑

```
let vm = new Vue({
el:"#app",
data:{},
router,
watch:{
$router(to,from){
console.log(to.path);
}
}
})
```

面试题 38. Vue不使用v-model的时候怎么监听数据变化?

推荐指数: ★★★★★ 试题难度: 中级 试题类型: 原理题

试题回答参考思路:

可以使用watch监听数据的变化

面试题 39. 简述Vue中同时发送多个请求怎么操作?

推荐指数: ★★★★ 试题难度: 初级 试题类型: 原理题

试题回答参考思路:

创建两个Promise，在Promise中使用axios
调用Promise.all([p1,p2]).then(res =>{ }).catch(err => { }) 方法
举例说明:

```
getInfo(){
//创建promise,在promise中调用axios ，then里使用resolve回调，catch里使用reject回调
```

```
var p1 = new Promie((resolve,reject) => {
this.$axios.get(httpUrl.getUser).then(res => {
resolve(res);
}).catch(err =>{
reject (err);
})
})

var p2 = new Promie((resolve,reject) => {
this.$axios.get(httpUrl.getCompany).then(res => {
resolve(res);
}).catch(err =>{
reject (err);
})
})
//调用Promise.add().then(res => {})
Promise.all([p1,p2]).then(res => {
console.log(res);
})
}
resolve(res);
```

面试题 40. 请说明Vue Watch和Dep的关系？

推荐指数：★★★★ 试题难度：初级 试题类型：原理题

试题回答参考思路：

watcher中实例化了dep并向dep.subs中添加了订阅者，dep通过notify遍历了dep.subs通知每个watcher更新。
依赖收集

initState时，对computed属性初始化时，触发computed watcher依赖收集
initState时，对侦听属性初始化是，触发user watcher依赖收集
render（）的过程， 触发render watcher依赖收集
re-render时，vm.render( )再次执行，会溢出所有subs中的watcher的订阅，重新赋值
派发更新
组件对响应的数据进行了修改，触发setter的逻辑
调用dep.notify( )
遍历所有的subs(Watcher实例),调用每一个watcher的update方法

面试题 41. 自定义指定(v-check/v-focus)的方法有哪些？有哪些钩子函数？还有哪些钩子函数参数？

推荐指数：★★★ 试题难度：中级 试题类型：原理题

试题回答参考思路：

全局定义指令：在vue对象的directive方法里面有两个参数，一个是指令名称，另外一个是函数。组件内定义指令：directives钩子函数：bind（绑定事件触发）、inserted（节点插入的时候触发）、updata（组件内相关更新）
钩子函数的参数：el、binding

面试题 42. 阐述Vue 中 computed 和 methods 的区别？

推荐指数：★★★★★ 试题难度：高难 试题类型：原理题

试题回答参考思路：

首先从表现形式上面来看， computed 和 methods 的区别大致有下面 4 点：

在使用时，computed 当做属性使用，而 methods 则当做方法调用
computed 可以具有 getter 和 setter，因此可以赋值，而 methods 不行
computed 无法接收多个参数，而 methods 可以
computed 具有缓存，而 methods 没有
而如果从底层来看的话， computed 和 methods 在底层实现上面还有很大的区别。

vue 对 methods 的处理比较简单，只需要遍历 methods 配置中的每个属性，将其对应的函数使用 bind 绑定当前组件实例后复制其引用到组件实例中即可

而 vue 对 computed 的处理会稍微复杂一些。

面试题 43. Vue 的数据为什么频繁变化但只会更新一次?

推荐指数：★★★★ 试题难度：中级 试题类型：原理题

试题回答参考思路：

这是因为 vue 的 DOM 更新是一个异步操作，在数据更新后会首先被 set 钩子监听到，但是不会马上执行 DOM 更新，而是在下一轮循环中执行更新。

具体实现是 vue 中实现了一个 queue 队列用于存放本次事件循环中的所有 watcher 更新，并且同一个 watcher 的更新只会被推入队列一次，并在本轮事件循环的微任务执行结束后执行此更新(UI Render 阶段)，这就是 DOM 只会更新一次的原因。

这种在缓冲时去除重复数据对于避免不必要的计算和 DOM 操作是非常重要的。然后，在下一个的事件循环“tick”中，vue 刷新队列并执行实际（已去重的）工作。vue 在内部对异步队列尝试使用原生的 Promise.then、MutationObserver 和 setImmediate，如果执行环境不支持，则会采用 setTimeout(fn, 0) 代替

面试题 44. 简述接口请求一般放在哪个生命周期中？为什么要这样做?

推荐指数：★★★ 试题难度：中级 试题类型：原理题

试题回答参考思路：

接口请求可以放在钩子函数 created、beforeMount、mounted 中进行调用，因为在这三个钩子函数中，data 已经创建，可以将服务端端返回的数据进行赋值。

但是推荐在 created 钩子函数中调用异步请求，因为在 created 钩子函数中调用异步请求有以下优点：

能更快获取到服务端数据，减少页面 loading 时间
SSR 不支持 beforeMount 、mounted 钩子函数，所以放在 created 中有助于代码的一致性
created 是在模板渲染成 html 前调用，即通常初始化某些属性值，然后再渲染成视图。如果在 mounted 钩子函数中请求数据可能导致页面闪屏问题

面试题 45. Vue 为什么没有类似于 React 中 shouldComponentUpdate 的生命周期?

推荐指数：★★★★ 试题难度：中级 试题类型：原理题

试题回答参考思路：

根本原因是 Vue 与 React 的变化侦测方式有所不同

React 是 pull 的方式侦测变化，当 React 知道发生变化后，会使用 Virtual Dom Diff 进行差异检测，但是很多组件实际上是肯定不会发生变化的，这个时候需要用 shouldComponentUpdate 进行手动操作来减少 diff，从而提高程序整体的性能。

Vue 是 pull+push 的方式侦测变化的，在一开始就知道那个组件发生了变化，因此在 push 的阶段并不需要手动控制 diff，而组件内部采用的 diff 方式实际上是可以引入类似于 shouldComponentUpdate 相关生命周期的，但是通常合理大小的组件不会有过量的 diff，手动优化的价值有限，因此目前 Vue 并没有考虑引入 shouldComponentUpdate 这种手动优化的生命周期

面试题 46. Vue.extend 和 Vue.component 的区别是什么?

推荐指数：★★★★★ 试题难度：中级 试题类型：原理题

试题回答参考思路：

Vue.extend 用于创建一个基于 Vue 构造函数的“子类”，其参数应为一个包含组件选项的对象。
Vue.component 用来注册全局组件。

面试题 47. 请说明scoped 是如何实现样式穿透的?

推荐指数：★★★★ 试题难度：高难 试题类型：原理题

试题回答参考思路：

首先说一下什么场景下需要 scoped 样式穿透。
在很多项目中，会出现这么一种情况，即：引用了第三方组件，需要在组件中局部修改第三方组件的样式，而又不想去除 scoped 属性造成组件之间的样式污染。此时只能通过特殊的方式，穿透 scoped。
有三种常用的方法来实现样式穿透。
方法一
使用 ::v-deep 操作符(>>> 的别名)
如果希望 scoped 样式中的一个选择器能够作用得“更深”，例如影响子组件，可以使用 >>> 操作符：

上述代码将会编译成：
.a[data-v-f3f3eg9] .b {
/* ... */
}
后面的类名没有 data 属性，所以能选到子组件里面的类名。
有些像 Sass 之类的预处理器无法正确解析 >>>，所以需要使用 ::v-deep 操作符来代替。
方法二
定义一个含有 scoped 属性的 style 标签之外，再定义一个不含有 scoped 属性的 style 标签，即在一个 vue 组件中定义一个全局的 style 标签，一个含有作用域的 style 标签：

此时，我们只需要将修改第三方样式的 css 写在第一个 style 中即可。
方法三
上面的方法一需要单独书写一个不含有 scoped 属性的 style 标签，可能会造成全局样式的污染。
更推荐的方式是在组件的外层 DOM 上添加唯一的 class 来区分不同组件，在书写样式时就可以正常针对针对这部分 DOM 书写样式。

面试题 48. Vue 如何快速定位那个组件出现性能问题的？

推荐指数：★★★★★ 试题难度：初级 试题类型：原理题

试题回答参考思路：

用 timeline 工具。 通过 timeline 来查看每个函数的调用时常，定位出哪个函数的问题，从而能判断哪个组件出了问题。

面试题 49. 简述Vue complier 的实现原理是什么样的?

推荐指数：★★★★★ 试题难度：中级 试题类型：原理题

试题回答参考思路：

在使用 vue 的时候，我们有两种方式来创建我们的 HTML 页面，第一种情况，也是大多情况下，我们会使用模板 template 的方式，因为这更易读易懂也是官方推荐的方法；第二种情况是使用 render 函数来生成 HTML，它比 template 更接近最终结果。

complier 的主要作用是解析模板，生成渲染模板的 render， 而 render 的作用主要是为了生成 VNode

complier 主要分为 3 大块：

parse：接受 template 原始模板，按着模板的节点和数据生成对应的 ast
optimize：遍历 ast 的每一个节点，标记静态节点，这样就知道哪部分不会变化，于是在页面需要更新时，通过 diff 减少去对比这部分DOM，提升性能
generate 把前两步生成完善的 ast，组成 render 字符串，然后将 render 字符串通过 new Function 的方式转换成渲染函数

面试题 50. Vue watch怎么深度监听对象变化 ？

推荐指数： ★★★★★ 试题难度： 高难 试题类型： 原理题

试题回答参考思路：

deep设置为true 就可以监听到对象的变化let vm=new Vue({el:"#first",data:{msg:{name:' 北 京 '}},watch:{msg:{handler (newMsg,oldMsg){console.log(newMsg);},immediate:true,deep:true}}})

面试题 51. Vue过渡动画实现的方式有哪些?

推荐指数： ★★★ 试题难度： 高难 试题类型： 原理题

试题回答参考思路：

1.使用vue的transition标签结合css样式完成动画
2.利用animate.css结合transition实现动画
3.利用 vue中的钩子函数实现动画

面试题 52. 简述Vue中如何扩展一个组件 ？

推荐指数： ★★★★★ 试题难度： 中级 试题类型： 原理题

试题回答参考思路：

常见的组件扩展方法有：mixins，slots，extends等

混入mixins是分发 Vue组件中可复用功能的非常灵活的方式。混入对象可以包含任意组件选项。当组件使用混入对象时，所有混入对象的选项将被混入该组件本身的选项。

插槽主要用于vue组件中的内容分发，也可以用于组件扩展。
子组件Child

这个内容会被父组件传递的内容替换

3
父组件Parent

来自老爹的内容

如果要精确分发到不同位置可以使用具名插槽，如果要使用子组件中的数据可以使用作用域插槽。

混入的数据和方法不能明确判断来源且可能和当前组件内变量产生命名冲突，vue3中引入的composition api，可以很好解决这些问题，利用独立出来的响应式模块可以很方便的编写独立逻辑并提供响应式的数据，然后在setup选项中组合使用，增强代码的可读性和维护性。例如：

```
// 复用逻辑1
function useXX() {}
// 复用逻辑2
function useYY() {}
// 逻辑组合
const Comp = {
setup() {
const {xx} = useXX()
const {yy} = useYY()
return {xx, yy}
}
}
```

面试题 53. 简述Vue自定义指令有哪些生命周期?

推荐指数：★★★★★ 试题难度：中级 试题类型：原理题

试题回答参考思路：

自定义指令的生命周期，有 5 个事件钩子，可以设置指令在某一个事件发生时的具体行为：

bind: 只调用一次，指令第一次绑定到元素时调用，用这个钩子函数可以定义一个在绑定时执行一次的初始化动作。
inserted: 被绑定元素插入父节点时调用（父节点存在即可调用，不必存在于 document 中）。
update: 被绑定元素所在的模板更新时调用，而不论绑定值是否变化。通过比较更新前后的绑定值，可以忽略不必要的模板更新（详细的钩子函数参数见下）。
componentUpdated: 被绑定元素所在模板完成一次更新周期时调用。
unbind: 只调用一次， 指令与元素解绑时调用。
钩子函数的参数（包括 el，binding，vnode，oldVnode）

el: 指令所绑定的元素，可以用来直接操作 DOM 。
binding: 一个对象，包含以下属性：name: 指令名、value: 指令的绑定值、oldValue: 指令绑定的前一个值、expression: 绑定值的字符串形式、arg: 传给指令的参数、modifiers: 一个包含修饰符的对象。

vnode: Vue 编译生成的虚拟节点。
oldVnode: 上一个虚拟节点，仅在 update 和 componentUpdated 钩子中可

面试题 54. 如何监听 pushstate 和 replacestate 的变化呢?

推荐指数: ★★★★ 试题难度: 高难 试题类型: 原理题

试题回答参考思路:

History.replaceState 和 pushState 不会触发 popstate 事件，所以我们可以通过在方法中创建一个新的全局事件来实现 pushstate 和 replacestate 变化的监听。

具体做法为:

```
var _wr = function(type) {
var orig = history[type];
return function() {
var rv = orig.apply(this, arguments);
var e = new Event(type);
e.arguments = arguments;
window.dispatchEvent(e);
return rv;
};
};
history.pushState = _wr('pushState');
history.replaceState = _wr('replaceState');
```

这样就创建了 2 个全新的事件，事件名为 pushState 和 replaceState，我们就可以在全局监听

```
window.addEventListener('replaceState', function(e) {
console.log('THEY DID IT AGAIN! replaceState 111111');
});
window.addEventListener('pushState', function(e) {
console.log('THEY DID IT AGAIN! pushState 2222222');
});
```

这样就可以监听到 pushState 和 replaceState 行为。

面试题 55. 请简述Vue 中相同逻辑如何进行抽离?

推荐指数: ★★★★ 试题难度: 中级 试题类型: 原理题

试题回答参考思路:

可以使用 vue 里面的混入（mixin）技术。混入（mixin）提供了一种非常灵活的方式，来将 vue 中相同的业务逻辑进行抽离。

例如:

在 data 中有很多是公用数据
引用封装好的组件也都是一样的
methods、watch、computed 中也都有大量的重复代码
当然这个时候可以将所有的代码重复去写来实现功能，但是我们并不不推荐使用这种方式，无论是工作量、工作效率和后期维护来说都是不建议的，这个时候 mixin 就可以大展身手了。

一个混入对象可以包含任意组件选项。当组件使用混入对象时，所有混入对象的选项将被“混合”进入该组件本身的选项。说白了就是给每个生命周期，函数等等中间加入一些公共逻辑。

混入技术特点

当组件和混入对象含有同名选项时，这些选项将以恰当的方式进行“合并”。比如，数据对象在内部会进行递归合并，并在发生冲突时以组件数据优先。
同名钩子函数将合并为一个数组，因此都将被调用。另外，混入对象的钩子将在组件自身钩子之前调用。
值为对象的选项，例如 methods、components 和 directives，将被合并为同一个对象。两个对象键名冲突时，取组件对象的键值对

面试题 56. 简述 Vue3.0 为什么速度更快？

推荐指数：★★★★★ 试题难度：中级 试题类型：原理题

试题回答参考思路：

优化 Diff 算法

相比 Vue 2，Vue 3 采用了更加优化的渲染策略。去掉不必要的虚拟 DOM 树遍历和属性比较，因为这在更新期间往往会产生最大的性能开销。

这里有三个主要的优化：

首先，在 DOM 树级别。
在没有动态改变节点结构的模板指令（例如 v-if 和 v-for）的情况下，节点结构保持完全静态。

当更新节点时，不再需要递归遍历 DOM 树。所有的动态绑定部分将在一个平面数组中跟踪。这种优化通过将需要执行的树遍历量减少一个数量级来规避虚拟 DOM 的大部分开销。

其次，编译器积极地检测模板中的静态节点、子树甚至数据对象，并在生成的代码中将它们提升到渲染函数之外。这样可以避免在每次渲染时重新创建这些对象，从而大大提高内存使用率并减少垃圾回收的频率。

第三，在元素级别。

编译器还根据需要执行的更新类型，为每个具有动态绑定的元素生成一个优化标志。

例如，具有动态类绑定和许多静态属性的元素将收到一个标志，提示只需要进行类检查。运行时将获取这些提示并采用专用的快速路径。

综合起来，这些技术大大改进了渲染更新基准，Vue 3.0 有时占用的 CPU 时间不到 Vue 2 的十分之一。

体积变小

重写后的 Vue 支持了 tree-shaking，像修剪树叶一样把不需要的东西给修剪掉，使 Vue 3.0 的体积更小。

需要的模块才会打入到包里，优化后的 Vue 3.0 的打包体积只有原来的一半（13kb）。哪怕把所有的功能都引入进来也只有 23kb，依然比 Vue 2.x 更小。像 keep-alive、transition 甚至 v-for 等功能都可以按需引入。

并且 Vue 3.0 优化了打包方法，使得打包后的 bundle 的体积也更小。

官方所给出的一份惊艳的数据：打包大小减少 41%，初次渲染快 55%，更新快 133%，内存使用减少 54%。

面试题 57. 动态给vue的data添加一个新的属性时会发生什么？怎样解决?

推荐指数：★★ 试题难度： 初级 试题类型： 原理题

试题回答参考思路：

如果在实例创建之后添加新的属性到实例上，它不会触发视图更新。如果想要使添加的值做到响应式，应当使用$set()来添加对象。

面试题 58. Vue中子组件可以直接改变父组件的数据么，说明原因？

推荐指数：★★★★★ 试题难度： 中级 试题类型： 原理题

试题回答参考思路：

所有的 prop 都使得其父子之间形成了一个单向下行绑定：父级 prop 的更新会向下流动到子组件中，但是反过来则不行。这样会防止从子组件意外变更父级组件的状态，从而导致你的应用的数据流向难以理解。另外，每次父级组件发生变更时，子组件中所有的 prop 都将会刷新为最新的值。这意味着你不应该在一个子组件内部改变 prop。如果你这样做了，Vue 会在浏览器控制台中发出警告。

```
const props = defineProps(['foo'])
// ❌ 下面行为会被警告，props是只读的!
props.foo = 'bar'
```

实际开发过程中有两个场景会想要修改一个属性：

这个 prop 用来传递一个初始值；这个子组件接下来希望将其作为一个本地的 prop 数据来

使用。在这种情况下，最好定义一个本地的 data，并将这个 prop 用作其初始值：
const props = defineProps(['initialCounter'])
const counter = ref(props.initialCounter)
这个 prop 以一种原始的值传入且需要进行转换。在这种情况下，最好使用这个 prop 的值来定义一个计算属性：
const props = defineProps(['size'])
// prop变化，计算属性自动更新
const normalizedSize = computed(() => props.size.trim().toLowerCase())
实践中如果确实想要改变父组件属性应该emit一个事件让父组件去做这个变更。注意虽然我们不能直接修改一个传入的对象或者数组类型的prop，但是我们还是能够直接改内嵌的对象或属性。

面试题 59. 解释Vue 插槽与作用域插槽的区别是什么?

推荐指数：★★★★★ 试题难度：初级 试题类型：原理题

试题回答参考思路：

插槽
创建组件虚拟节点时，会将组件儿子的虚拟节点保存起来。当初始化组件时，通过插槽属性将儿子进行分类 {a:[vnode],b[vnode]}
渲染组件时会拿对应的 slot 属性的节点进行替换操作。（插槽的作用域为父组件）

作用域插槽
作用域插槽在解析的时候不会作为组件的孩子节点。会解析成函数，当子组件渲染时，会调用此函数进行渲染。
普通插槽渲染的作用域是父组件，作用域插槽的渲染作用域是当前子组件。

面试题 60. Vue生命周期钩子是如何实现的?

推荐指数：★★★★ 试题难度：中级 试题类型：原理题

试题回答参考思路：

生命周期钩子在内部会被vue维护成一个数组(vue 内部有一个方法mergeOption)和全局的生命周期合并最终转换成数组，当执行到具体流程时会执行钩子(发布订阅模式)，callHook来实现调用。

理解vue中模板编译原理?

1 会将模板变成ast语法树(模板编译原理的核心就是 ast-> 生成代码)
2 对 ast 语法树进行优化，标记静态节点.(vue3 中模板编译做了哪些优化 patchFlag,blockTree,事件缓存，节点缓存...)
3 代码生成 拼接render函数字符串 + new Function + with;

面试题 61. 如何理解Vue中的模板编译原理?

推荐指数：★★★★★ 试题难度：中级 试题类型：原理题 ▸

试题回答参考思路：

关于Vue编译原理这块的整体逻辑主要分为三步：

第一步将模版字符串转换成element ASTs(解析器)
第二步是对AST进行静态节点标记，主要用来做虚拟DOM的渲染优化(优化器)
第三步是使用element ASTs生成render函数代码字符串(代码生成器)
解析器

{{name}}

上面一个简单 的模版转换成element AST树形结构后是这样的：

```
{
tag: "div"
type: 1,
staticRoot: false,
static: false,
plain: true,
parent: undefined,
attrsList: [],
attrsMap: {},
children: [
{
tag: "p"
type: 1,
staticRoot: false,
static: false,
plain: true,
parent: {tag: "div", ...},
attrsList: [],
attrsMap: {},
children: [{
type: 2,
text: "{{name}}",
static: false,
expression: "_s(name)"
}]
}
]
}
```

我们可以看到上面的dom被解析成了解析器，它的原理主要是两部分内容，一部分是截取字符串，一部分是对截取的字符串做解析。

优化器
优化器的目的就是找出那些静态节点并打上标记，而静态节点指的是DOM不需要发生变化的

节点，也就是里面都是静态标签和静态文本。

标记静态节点有两个好处：
一、每次重新渲染的时候不需要为静态节点创建新节点，也就是静态节点的解析器不需要重新创建
二、在Virtual DOM中patching的过程可以被跳过
优化器的实现原理主要分两步：
一、用递归的方式将静态节点添加static属性，用来标识是不是静态节点
二、标记所有静态根节点(子节点全是静态节点就是静态根节点)
代码生成器
代码生成器的作用是使用element ASTs生成render函数代码字符串。
使用本文开头举的例子中的模版生成后的AST来生成render后是这样的：

```
{
render: with(this){return _c('div',[_c('p',[_v(_s(name))])])}
}
```

格式化后是这样的：

```
with(this){
return _c(
'div',
[
_c(
'p',
[
_v(_s(name))
]
)
]
)
}
```

生成后的代码字符串中看到了有几个函数调用_c、_v、_s。
_c对应的是createElement，它的作用是创建一个元素。
1.第一个参数是一个HTML标签名
2.第二个参数是元素上使用的属性所对应的数据对象，可选项
3.第三个参数是children
_v的意思是创建一个文本节点。
_s是返回参数中的字符串。
代码生成器的总体逻辑其实就是使用element ASTs去递归，然后拼出这样的_c('div',[_c('p',[_v(_s(name))])]) 字符串。

总结
本篇文章我们说了 vue 对模板编译的整体流程分为三个部分：解析器（parser），优化器（optimizer）和代码生成器（code generator）。
解析器（parser）的作用是将 模板字符串 转换成 element ASTs。
优化器（optimizer）的作用是找出那些静态节点和静态根节点并打上标记。
代码生成器（code generator）的作用是使用 element ASTs 生成 render函数代码

（generate render function code from element ASTs）。

解析器（parser）的原理是一小段一小段的去截取字符串，然后维护一个 stack 用来保存DOM深度，每截取到一段标签的开始就 push 到 stack 中，当所有字符串都截取完之后也就解析出了一个完整的 AST。

优化器（optimizer）的原理是用递归的方式将所有节点打标记，表示是否是一个 静态节点，然后再次递归一遍把 静态根节点 也标记出来。

代码生成器（code generator）的原理也是通过递归去拼一个函数执行代码的字符串，递归的过程根据不同的节点类型调用不同的生成方法，如果发现是一颗元素节点就拼一个 _c(tagName, data, children) 的函数调用字符串，然后 data 和 children 也是使用 AST 中的属性去拼字符串。

如果 children 中还有 children 则递归去拼。

最后拼出一个完整的 render 函数代码。

面试题 62. 请简述vue-router 动态路由是什么?

推荐指数: ★★★★ 试题难度: 初级 试题类型: 原理题

试题回答参考思路:

我们经常需要把某种模式匹配到的所有路由，全都映射到同个组件。例如，我们有一个 User 组件，对于所有 ID 各不相同的用户，都要使用这个组件来渲染。那么，我们可以在 vue-router 的路由路径中使用“ 动态路径参数 ”（ dynamic segment ）来达到这个效果:

```
const User = {
template: "
User", };
const router = new VueRouter({
routes: [
// 动态路径参数 以冒号开头
{ path: "/user/:id", component: User },
],
});
```

面试题 63. 简述 Vue 2.0 响应式数据的原理（ 重点 ）?

推荐指数: ★★★★★ 试题难度: 高难 试题类型: 原理题

试题回答参考思路:

整体思路是 数据劫持 + 观察者模式
Vue 在初始化数据时 ， 会使用 Object.defineProperty 重新定义 data 中的所有属性 ，

当页面 使用对 应 属性时，首先会进行 依赖收集（收集当前组件的 watcher ），如果属性发生变化 会通知相关 依赖进行 更新操作(发布订阅)
Vue2.x 采用 数据劫持结合发布订阅模式 （PubSub 模式）的方式，通过Object.defineProperty 来劫 持 各个属性 的 setter、getter ，在 数据变动时 发 布消息给订阅者 ， 触发相应的监听回 调。
当把一个普通 Javascript 对象传给 Vue 实例来作为它的 data 选项时，Vue 将遍历它的属性，用
Object.defineProperty 将它们转为 getter/setter。用户看不到 getter/setter，但是在内部它们让
Vue 追踪依赖，在属性被访问和修改时 通知变化 。
Vue 的数据 双向绑定 整合了 Observer，Compile 和 Watcher 三者，通过 Observer 来监听 自己的
model 的数据变化，通过 Compile 来解析编 译模板指令，最终 利用 Watcher 搭 起 Observer 和
Compile 之间的 通信桥梁 ，达到数据变化->视图更新，视图交互变化（例如 input 操作）->数据
model 变更的双向绑定效果。
Vue3.x 放弃了 Object.defineProperty ，使用 ES6 原生的 Proxy，来解决以前使用 Object.defineProperty 所存在的一些问题。
1、Object.defineProperty 数据劫持
2、使用 getter 收集依赖 ，setter 通知 watcher派发更新。
3、watcher 发布订阅模式。

面试题 64. 简述active-class是哪个组件的属性？嵌套路由怎么定义？

推荐指数：★★★★★ 试题难度： 初级 试题类型： 原理题

试题回答参考思路：

vue-router模块的router-link组件。

面试题 65. 简述scss是什么？在vue.cli中的安装使用步骤是？有哪几大特性？

推荐指数：★★ 试题难度： 初级 试题类型： 原理题

试题回答参考思路：

css的预编译。
使用步骤：
第一步：用npm 下三个loader（sass-loader、css-loader、node-sass）
第二步：在build目录找到webpack.base.config.js，在那个extends属性中加一个拓展.scss
第三步：还是在同一个文件，配置一个module属性
第四步：然后在组件的style标签加上lang属性 ，例如：lang=”scss”
有哪几大特性:
1、可以用变量，例如（$变量名称=值）；

2、可以用混合器，例如（）
3、可以嵌套

面试题 66. 简述mint-ui是什么？怎么使用？说出至少三个组件使用方法？

推荐指数：★★★★★ 试题难度：初级 试题类型：原理题

试题回答参考思路：

基于vue的前端组件库。npm安装，然后import样式和js，vue.use（mintUi）全局引入。在单个组件局
部引入：import {Toast} from 'mint-ui'。组件一：Toast('登录成功')；组件二：mint-header；组件三：
mint-swiper

面试题 67. 简述Vue.js的template编译的理解？

推荐指数：★★★★ 试题难度：中级 试题类型：原理题

试题回答参考思路：

简而言之，就是先转化成AST树，再得到的render函数返回VNode（Vue的虚拟DOM节点）
详情步骤：
首先，通过compile编译器把template编译成AST语法树（abstract syntax tree 即 源代码的抽象语法结构的树状表现形式），compile是createCompiler的返回值，createCompiler是用以创建编译器的。另外compile还负责合并option。
然后，AST会经过generate（将AST语法树转化成render funtion字符串的过程）得到render函数，render的返回值是VNode，VNode是Vue的虚拟DOM节点，里面有（标签名、子节点、文本等等）

面试题 68. 简述Vue声明组件的state是用data方法，那为什么data是通过一个function来返 回一个对象，而不是直接写一个对象呢?

推荐指数：★★★★ 试题难度：中级 试题类型：原理题

试题回答参考思路：

从语法上说，如果不用function返回就会出现语法错误导致编译不通过。从原理上的话，大概就是组件可以被多次创建，如果不使用function就会使所有调用该组件的页面公用同一个数据域，这样就失去了组件的概念了

面试题 69. 简述Vue中mixin与extend区别？

推荐指数：★★★ 试题难度：中级 试题类型：原理题

试题回答参考思路：

全局注册混合对象，会影响到所有之后创建的vue实例，而Vue.extend是对单个实例进行扩展。
mixin 混合对象（组件复用）
同名钩子函数（bind，inserted，update，componentUpdate，unbind）将混合为一个数组，因此都将被调用，混合对象的钩子将在组件自身钩子之前调用methods，components，directives将被混为同一个对象。两个对象的键名（方法名，属性名）冲突时，取组件（而非mixin）对象的键值对。

面试题 70. 简述prop 如何指定其类型要求？

推荐指数：★★★ 试题难度：初级 试题类型：原理题

试题回答参考思路：

通过实现 prop 验证选项，可以单个 prop 指定类型要求。这对生产没有影响，但是会在开发段发出警告，从而帮助开发人员识
别传人数据和 prop 的特定类型要求的潜在问题。
配置三个 prop 的例子 ：

```
props : {
accountNumber:{
type : Number,
required : true
},
name :{
type : String,
required : true
},
favoriteColors : Array
}
```

面试题 71. 简述什么是 mixin？

推荐指数：★★★★★ 试题难度：初级 试题类型：原理题

试题回答参考思路：

Mixin 使我们能够为 Vue 组件编写可插拔和可重用的功能。 如果你希望再多个组件之间重用一组组件选项，例如生命周期hook、 方法等，则可以将其编写为 mixin，并在组件中简单的引用它。然后将 mixin 的内容合并到组件中。如果你要在 mixin中定义生命周期 hook，那么它在执行时将优化于组件自己的 hook。

面试题 72. 简述什么是Vue渲染函数？举个例子？

推荐指数：★★★★ 试题难度：初级 试题类型：原理题

试题回答参考思路：

Vue 允许我们以多种方式构建模板，其中最常见的方式是只把 HTML 与特殊指令和 mustache 标签一起用于相响应功能。但是
你也可以通过 JavaScript 使用特殊的函数类 （称为渲染函数） 来构建模板。这些函数与编译器非常接近，这意味它们比其他
模板类型更高效、快捷。由于你使用 JavaScript 编写渲染函数，因此可以在需要的地方自由使用该语言直接添加自定义函
数。
对于标准 HTML 模板的高级方案非常有用。
这里是用 HTML 作为模板 Vue 程序

```
new Vue ({
el: '#app',
data:{
fruits: ['Apples','Oranges','Kiwi'] },
template:
`

# Fruit Basket

    1. {{ fruit }}

`
});
```

这里是用渲染函数开发的同一个程序：

```
new Vue({
el: '#app',
data: {
fruits: ['Apples', 'Oranges', 'Kiwi'] },
render: function(createElement) {
return createElement('div', [
createElement('h1', 'Fruit Basket'),
createElement('ol', this.fruits.map(function(fruit) {
return createElement('li', fruit);
}))
]);
}
});
```

输出如下：
Fruit Basket
1、Apples 2、Oranges 3、Kiwi
在上面的例子中，我们用了一个函数，它返回一系列 createElement() 调用，每个调用负责生成一个元素。尽管 v-for 指令在

基于 HTML 的模板中起作用，但是当时用渲染函数时，可以简单的用标准的 .map() 函数遍历 fruits 数据数组。

面试题 73. 简述什么时候调用“updated”生命周期 hook？ ？

推荐指数：★★★★ 试题难度： 初级 试题类型： 原理题

试题回答参考思路：

在更新响应性数据并重新渲染虚拟 DOM 之后，将调用更新的 hook。它可以用于执行与 DOM 相关的操作，但是（默认情况下）不能保证子组件会被渲染，尽管也可以通过在更新函数中使用 this.$nextTick 来确保。

面试题 74. 简述在 Vue 实例中编写生命周期 hook 或其他 option/propertie 时，为什么不使用箭头函数？

推荐指数：★★★ 试题难度： 初级 试题类型： 原理题

试题回答参考思路：

箭头函数自已没有定义 this 上下文中。当你在 Vue 程序中使用箭头函数（ => ）时，this 关键字病不会绑定到 Vue 实例，因此会引发错误。所以强烈建议改用标准函数声明

面试题 75. 简述Vue中引入组件的步骤？

推荐指数：★★ 试题难度： 初级 试题类型： 原理题

试题回答参考思路：

1.采用ES6的import ... from ...语法或CommonJS的require()方法引入组件
2.对组件进行注册,代码如下
// 注册
Vue.component('my-component', {
template: '

A custom component!
'
})
3.使用组件
子组件需要数据，可以在props中接受定义。而子组件修改好数据后，想把数据传递给父组件。可以采用emit方法。

面试题 76. 简述如何让CSS只在当前组件中起作用？

推荐指数：★★★★★ 试题难度： 初级 试题类型： 原理题

试题回答参考思路：

将当前组件的