38 Vue 优质真题深度解读

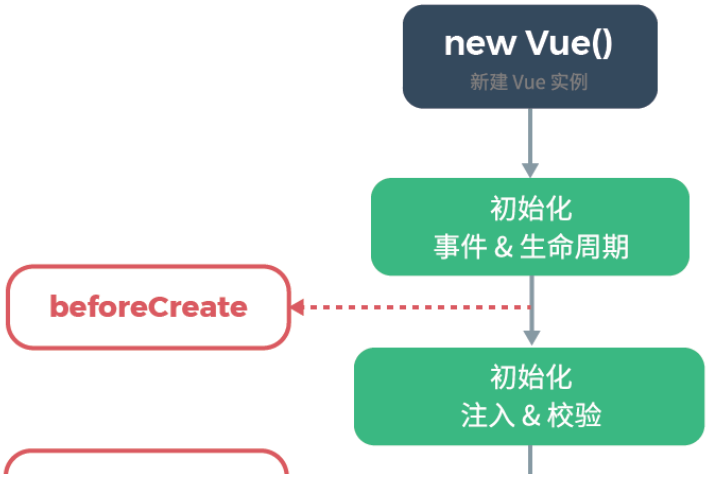
更新时间：2020-06-04 09:30:09

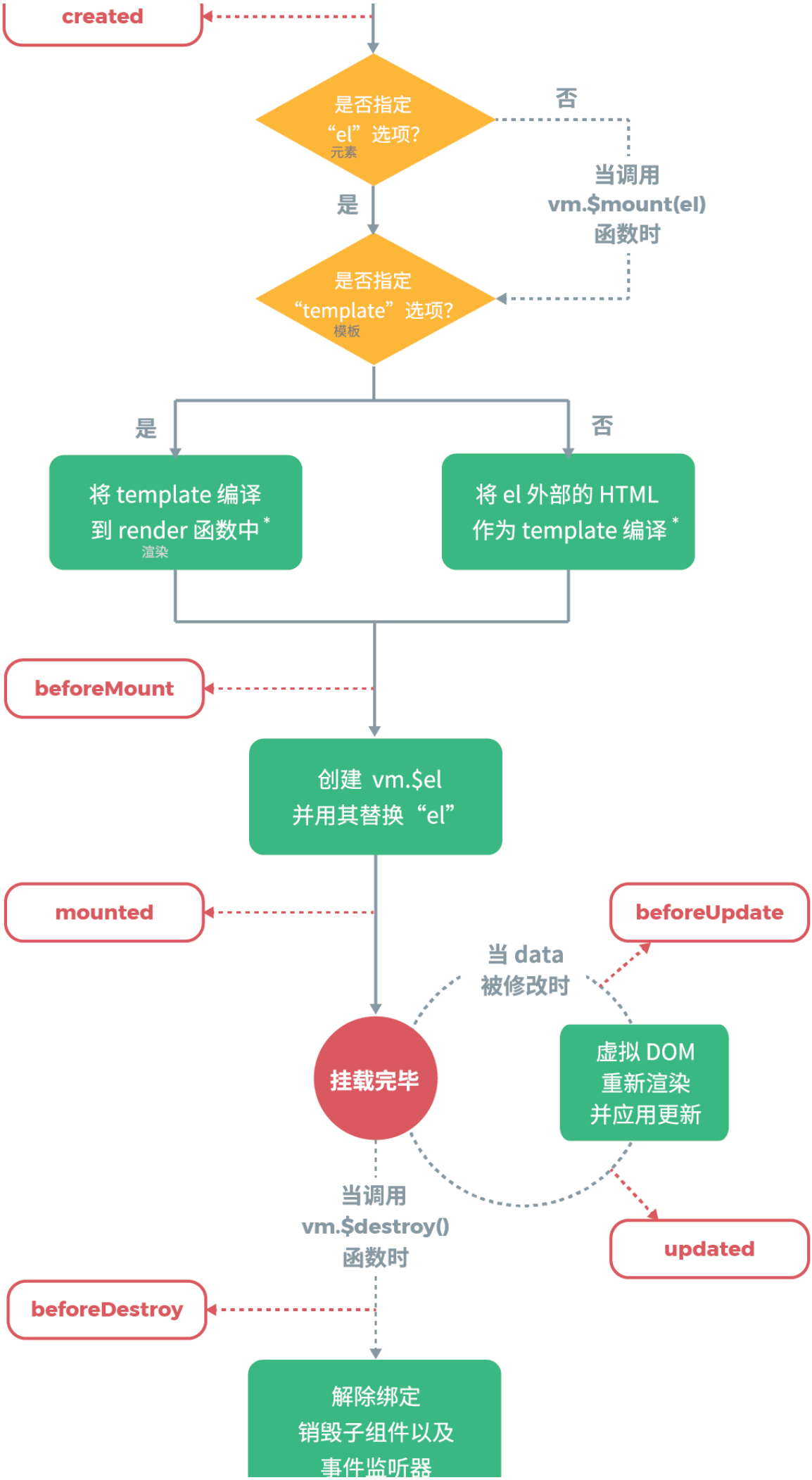
> 不想当将军的士兵，不是好士兵。——拿破仑

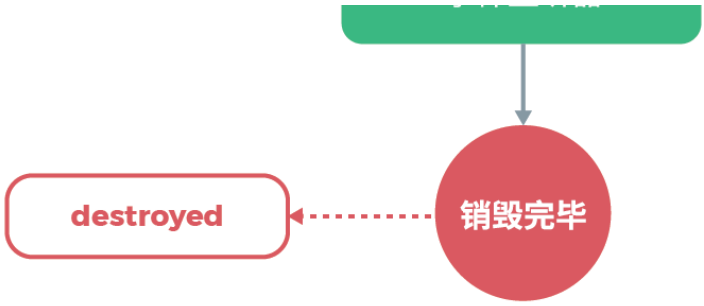
本节我们给大家补充一些 Vue 相关的经典面试真题。

大家知道，Vue 的面试题要真是收集起来，那可真够五花八门。本节涉及的题目可谓“沙里淘金”，同时具备了“频率高”、“区分度高”两个特性，普遍具有稳定的话题热度，能够确实反映一些知识结构、知识深度上问题。

请尝试描述 Vue 的生命周期



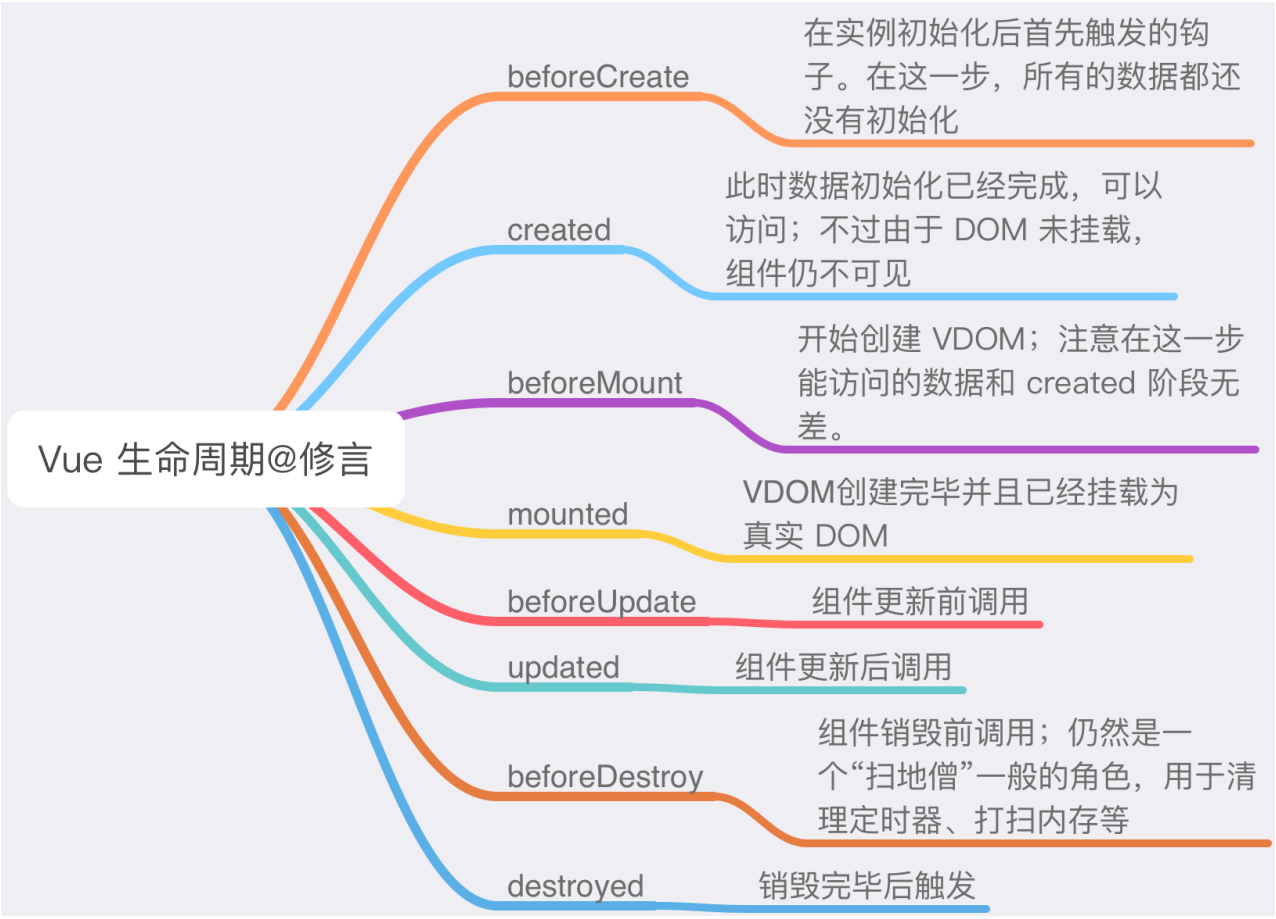

created
是否指定
“el”选项?
元素
否
当调用
vm.$mount(el)
函数时
是
是否指定
“template”选项?
模板
是
否
将 template 编译
到 render 函数中*
渲染
将 el 外部的 HTML
作为 template 编译*
beforeMount
创建 vm.$el
并用其替换“el”
mounted
beforeUpdate
当 data
被修改时
挂载完毕
虚拟 DOM
重新渲染
并应用更新
当调用
vm.$destroy()
函数时
updated
beforeDestroy
解除绑定
销毁子组件以及
事件监听器



* 如果使用构造生成文件（例如构造单文件组件），
模板编译将提前执行

（图源 Vue 官网）

生命周期，就是指实例从创建到销毁的过程。这里我制作了一张思维导图来帮助大家记忆每个环节的特性：



keep-alive 与生命周期

keep-alive是 Vue 提供的一个内置组件，用来对组件进行缓存——在组件切换过程中将状态保留在内存中，防止重复渲染DOM。

如果你为一个组件包裹了 keep-alive，那么它会多出两个生命周期：deactivated、actived。同时，beforeDestroy 和 destroyed 就不会再被触发了——毕竟组件不会被真正销毁。

当组件被换掉时，会被缓存到内存中、触发 deactivated 生命周期；当组件被切回来时，再去缓存里找这个组件、触发 actived。

Vue 的路由有哪些模式？说说你对前端路由的理解？

熟悉 vue-router 的同学都知道，Vue 路由有三种模式：hash、history 和 abstract：

- hash: 使用 URL hash 值来作路由。支持所有浏览器；
- history : 需要 HTML5 History API 和服务器配置结合。对浏览器版本有要求，不支持低版本浏览器；
- abstract : 支持所有 JavaScript 运行环境。如果当前环境没有浏览器 API，路由会自动进入这个模式.

这里我们需要关注的是前两种。其实这种机制并非 Vue 独有，它来源于现在业界广为大家接受的前端路由方案思路。当面试官问你这个问题的时候，他想听到的肯定不只是一个单薄的“hash and history”，而是希望挖掘你对 SPA 局限性、前端路由实现原理的理解。

理解前端路由——是什么？解决什么问题？

背景——问题的产生

在前端技术早期，一个 url 对应一个页面，如果你要从 A 页面切换到 B 页面，那么必然伴随着页面的刷新。

这个体验并不好，不过在最初也是无奈之举——用户只有在刷新页面的情况下，才可以重新去请求数据。

后来，改变发生了——Ajax 出现了，它允许人们在不刷新页面的情况下发起请求；与之共生的，还有“不刷新页面即可更新页面内容”这种需求。在这样的背景下，出现了 **SPA（单页面应用）**。

SPA极大地提升了用户体验，它允许页面在不刷新的情况下更新页面内容，使内容的切换更加流畅。但是在 SPA 诞生之初，人们并没有考虑到“定位”这个问题——在内容切换前后，页面的 URL 都是一样的，这就带来了两个问题：

- SPA 其实并不知道当前的页面“进展到了哪一步”。可能你在一个站点下经过了反复的“前进”才终于唤出了某一块内容，但是此时只要刷新一下页面，一切就会被清零，你必须重复之前的操作、才可以重新对内容进行定位——SPA 并不会“记住”你的操作。
- 由于有且仅有一个 URL 给页面做映射，这对 SEO 也不够友好，搜索引擎无法收集全面的信息

为了解决这个问题，前端路由出现了。

前端路由——SPA“定位”解决方案

前端路由可以帮助我们在仅有一个页面的情况下，“记住”用户当前走到了哪一步——为 SPA 中的各个视图匹配一个唯一标识。这意味着用户前进、后退触发的新内容，都会映射到不同的 URL 上去。此时即便他刷新页面，因为当前的 URL 可以标识出他所处的位置，因此内容也不会丢失。

那么如何实现这个目的呢？首先我们要解决两个问题：

- 当用户刷新页面时，浏览器会默认根据当前 URL 对资源进行重新定位（发送请求）。这个动作对 SPA 是不必要的，因为我们的 SPA 作为单页面，无论如何也只会有一个资源与之对应。此时若走正常的请求-刷新流程，

反而会使用户的前进后退操作无法被记录。

- 单页面应用对服务端来说，就是一个URL、一套资源，那么如何做到用“不同的URL”来映射不同的视图内容呢？

从这两个问题来看，服务端已经完全救不了这个场景了。所以要靠咱们前端自力更生，不然怎么叫“前端路由”呢？作为前端，我们可以提供这样的解决思路：

- 拦截用户的刷新操作，避免服务端盲目响应、返回不符合预期的资源内容。把刷新这个动作完全放到前端逻辑里消化掉。
- 感知 URL 的变化。这里不是说要改造 URL、凭空制造出 N 个 URL 来。而是说 URL 还是那个 URL，只不过我们可以给它做一些微小的处理——这些处理并不会影响 URL 本身的性质，不会影响服务器对它的识别，只有我们前端感知的到。一旦我们感知到了，我们就根据这些变化、用 JS 去给它生成不同的内容。

实践思路——hash 与 history

接下来重点就来了，现在前端界对前端路由有哪些实现思路？这里大家需要掌握的两个实践就是 hash 与 history。

hash 模式

hash 模式是指通过改变 URL 后面以“#”分隔的字符串（这货其实就是 URL 上的哈希值）、从而让页面感知到路由变化的一种实现方式。举个例子，比如这样的一个 URL ：

```
https://www.imooc.com/
```

我就可以通过增加和改变哈希值，来让这个 URL 变得有那么一点点不一样：

```
// 主页
https://www.imooc.com/#index

// 活动页
https://www.imooc.com/#activePage
```

这个“不一样”是我们前端完全可感知的——JS 可以帮我们捕获到哈希值的内容。在 hash 模式下，我们实现路由的思路可以概括如下：

1. hash 的改变：我们可以通过 location 暴露出来的属性，直接去修改当前 URL 的 hash 值：

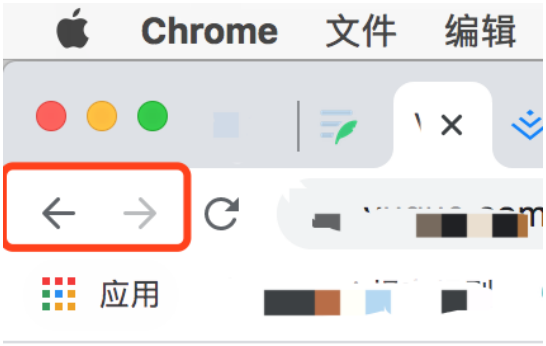
```
window.location.hash = 'index';
```

1. hash 的感知：通过监听 “hashchange”事件，我们可以用 JS 来捕捉 hash 值的变化，进而决定我们页面内容是否需要更新：

```
// 监听hash变化，点击浏览器的前进后退会触发
window.addEventListener('hashchange', function(event){
   // 根据 hash 的变化更新内容
},false)
```

history 模式

大家知道，在我们浏览器的左上角，往往有这样的操作点：



通过点击前进后退箭头，我们可以实现页面间的跳转。这样的行为，其实是可以通过 API 来实现的。

浏览器的 history API 赋予了我们这样的能力，在 HTML4 时，我们就可以通过下面的接口来操作浏览历史、实现跳转动作：

```
window.history.forward()  // 前进到下一页
```

```
window.history.back() // 后退到上一页
```

```
window.history.go(2) // 前进两页
```

```
window.history.go(-2) // 后退两页
```

很有趣吧？遗憾的是，在这个阶段，我们能做的只是“切换”，而不能“改变”。好在从 HTML5 开始，浏览器支持了 pushState 和 replaceState 两个 API，允许我们对浏览历史进行修改和新增：

```
history.pushState(data[,title][,url]); // 向浏览历史中追加一条记录
```

```
history.replaceState(data[,title][,url]); // 修改（替换）当前页在浏览历史中的信息
```

这样一来，修改动作就齐活了了。

有修改，就要有对修改的感知能力。在 history 模式下，我们可以通过监听 popstate 事件来达到我们的目的：

```
window.addEventListener('popstate', function(e) {
  console.log(e)
});
```

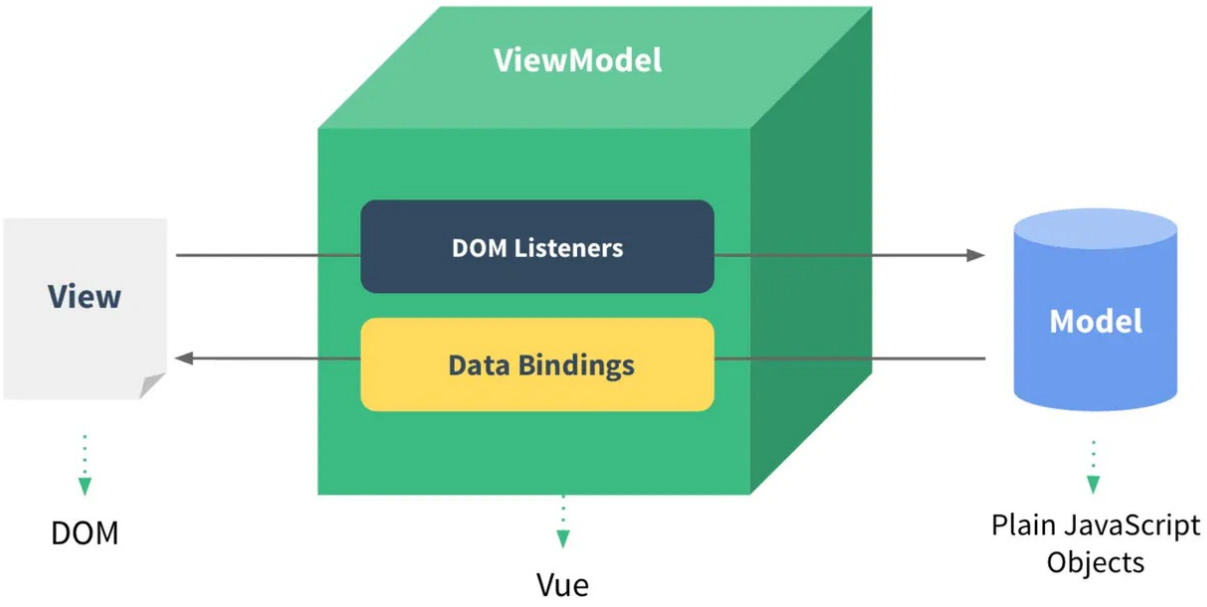
每当浏览历史发生变化，popstate 事件都会被触发。**注意这里有个坑**：go、forward 和 back 等方法的调用确实会触发 popstate，但是 **pushState** 和 **replaceState** 不会。

这个缺憾问题也不大，我们一般通过自定义事件（可复习我们前面的事件专题）和全局事件总线（后续设计模式专题会讲解）来实现。

注意：上述 history API 均不会导致页面的刷新。

说说你对 MVVM 的理解

mvvm 是一种架构模式。传统MVC导致的前端逻辑冗余的问题，本质原因是Model -> View可以直接通信，导致前端承载了过多的Model加工处理逻辑。在 mvvm 架构模式下，不存在这种问题：



这是 Vue 的 mvvm 示意图，其中，这三个角色大家要认清楚：

- View 层：视图层，对应到 <template> 标签的内容。
- VM 层：View-Model，对应到 Vue 实例。这一层是 View 和 Model 间的媒介。当用户操作通过 View 修改 View-Model 层的数据后，View-Model 会去修改 Model，然后再反过来把修改后的数据映射到 View 层上去。
- Model 层： 模型层，其实就是数据层。它对应到 Vue 中的数据。这个数据并非一个固定的实体，它可以代指 data 属性，也可以代指 Vuex 提供的数据，总之，它是页面所依赖的 JS 数据对象。

MVVM模型的关键，在于View的变化会直接映射在 ViewModel 中（反之亦然）。这个映射的过程不需要你手动实现，而是 MVVM 框架来帮你做掉。

这样一来，开发者开发 View 中的显示逻辑和 View-Model 中调用model的业务逻辑可以隔离的非常好，不需要在 View 中还去维护一块和 ViewModel 间的逻辑。

“隔离”意味着什么？没错，就是解耦合！不同的人写不同的代码，彼此井水不犯河水。进而达到的就是关注点分离——负责业务逻辑的开发人员不需要关心显示细节，负责显示逻辑的人不需要关心业务逻辑细节，项目复杂度由此大大降低。

}

37 Vue核心——nextTick原理源码级解析

39 TCP 与 UDP