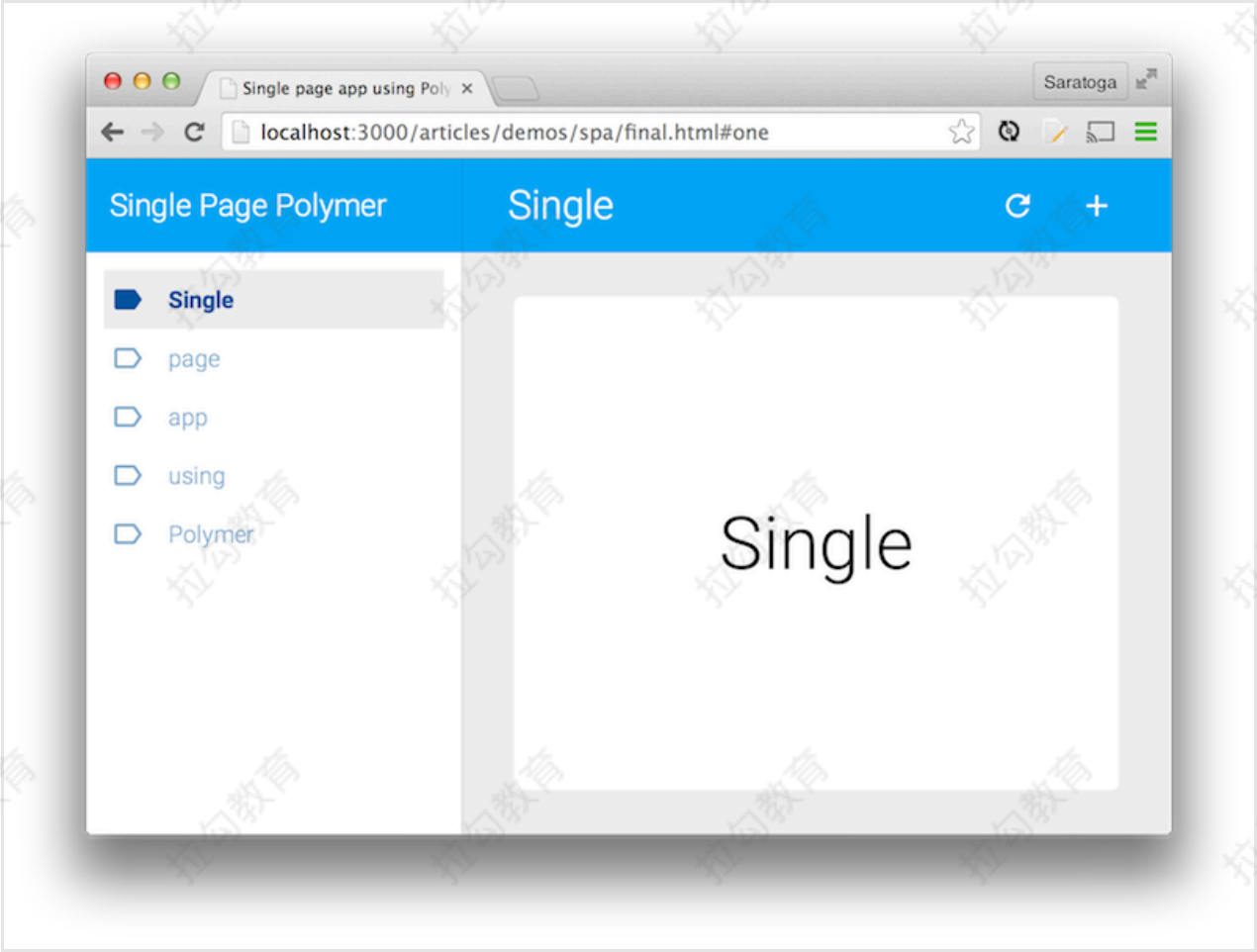
深度剖析 Vue.js 经典面试题

1. 谈一谈你对 SPA 单页面的理解，它的优缺点分别是什么



SPA（ single-page application ）仅在 Web 页面初始化时加载相应的 HTML、JavaScript 和 CSS。一旦页面加载完成，SPA 不会因为用户的操作而进行页面的重新加载或跳转；取而代之的是利用路由机制实现 HTML 内容的变换，UI 与用户的交互，避免页面的重新加载。

优点：

- 用户体验好、快，内容的改变不需要重新加载整个页面，避免了不必要的跳转和重复渲染；
- 基于上面一点，SPA 相对对服务器压力小；
- 前后端职责分离，架构清晰，前端进行交互逻辑，后端负责数据处理；

缺点：

- 首屏（初次）加载慢：为实现单页 Web 应用功能及显示效果，需要在加载页面的时候将 JavaScript、CSS 统一加载，部分页面按需加载；
- 不利于 SEO：由于所有的内容都在一个页面中动态替换显示，所以在 SEO 上其有着天然的弱势。

2. SPA 单页面应用的实现方式有哪些

> 前端路由的实现原理。

- hash 模式
- history 模式

在 `hash` 模式中，在 `window` 上监听 `hashchange` 事件（地址栏中hash变化触发）驱动界面变化；

在 `history` 模式中，在 `window` 上监听 `popstate` 事件（浏览器的前进或后退按钮的点击触发）驱动界面变化，监听 a 链接点击事件用 `history.pushState`、`history.replaceState`方法驱动界面变化；

直接在界面用显示隐藏事件驱动界面变化。

3. 使用过 Vue SSR 吗？说说 SSR?

> Server Side Renderer
>
> Vue.js 是构建客户端应用程序的框架。默认情况下，可以在浏览器中输出 Vue 组件，进行生成 DOM 和操作 DOM。然而，也可以将同一个组件渲染为服务端的 HTML 字符串，将它们直接发送到浏览器，最后将这些静态标记"激活"为客户端上完全可交互的应用程序。
>
> 即：SSR大致的意思就是vue在客户端将标签渲染成的整个 html 片段的工作在服务端完成，服务端形成的html 片段直接返回给客户端这个过程就叫做服务端渲染。

服务端渲染 SSR 的优缺点如下：

（1）服务端渲染的优点：

- 更好的 SEO： 因为 SPA 页面的内容是通过 Ajax 获取，而搜索引擎爬取工具并不会等待 Ajax 异步完成后再抓取页面内容，所以在 SPA 中是抓取不到页面通过 Ajax 获取到的内容；而 SSR 是直接由服务端返回已经渲染好的页面（数据已经包含在页面中），所以搜索引擎爬取工具可以抓取渲染好的页面；
- 更快的内容到达时间（首屏加载更快）： SPA 会等待所有 Vue 编译后的 js 文件都下载完成后，才开始进行页面的渲染，文件下载等需要一定的时间等，所以首屏渲染需要一定的时间；SSR 直接由服务端渲染好页面直接返回显示，无需等待下载 js 文件及再去渲染等，所以 SSR 有更快的内容到达时间；

(2) 服务端渲染的缺点：

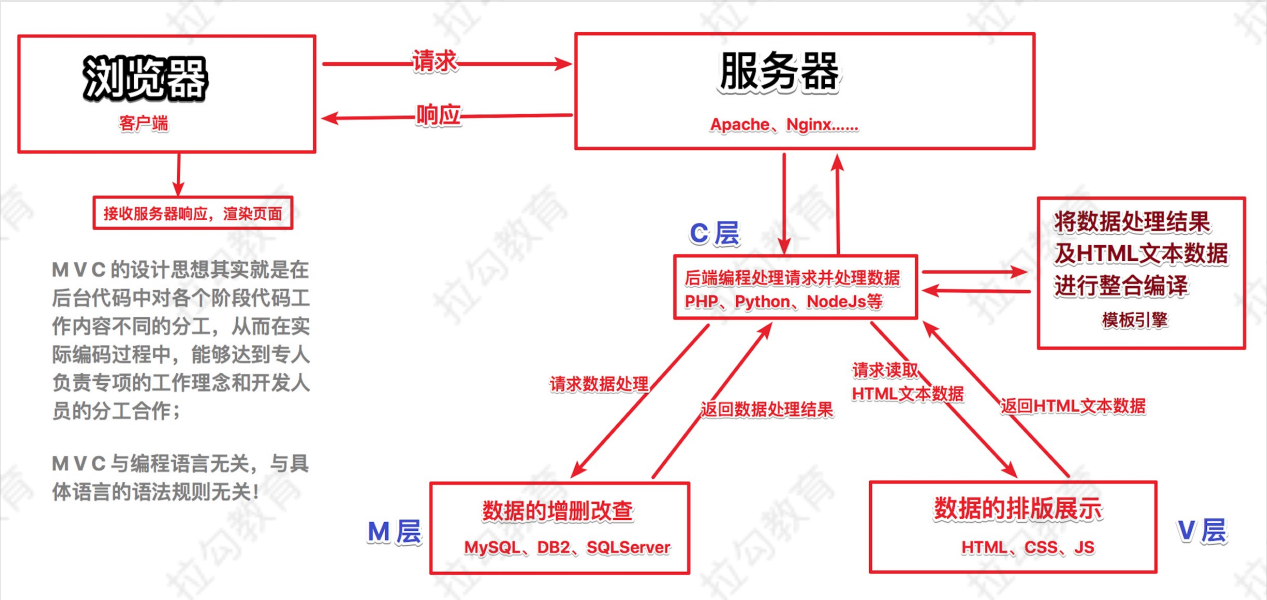
- 更多的开发条件限制： 例如服务端渲染只支持 beforCreate 和 created 两个钩子函数，这会导致一些外部扩展库需要特殊处理，才能在服务端渲染应用程序中运行；并且与可以部署在任何静态文件服务器上的完全静态单页面应用程序 SPA 不同，服务端渲染应用程序，需要处于 Node.js server 运行环境；
- 更多的服务器负载：在 Node.js 中渲染完整的应用程序，显然会比仅仅提供静态文件的 server 更加大量占用CPU 资源 (CPU-intensive - CPU 密集)，因此如果你预料在高流量环境 (high traffic) 下使用，请准备相应的服务器负载，并明智地采用缓存策略。

如果没有 SSR 开发经验的同学，可以参考本文作者的另一篇 SSR 的实践文章《Vue SSR 踩坑之旅》，里面 SSR 项目搭建以及附有项目源码。

4. 谈一谈你对 MVVM 的理解

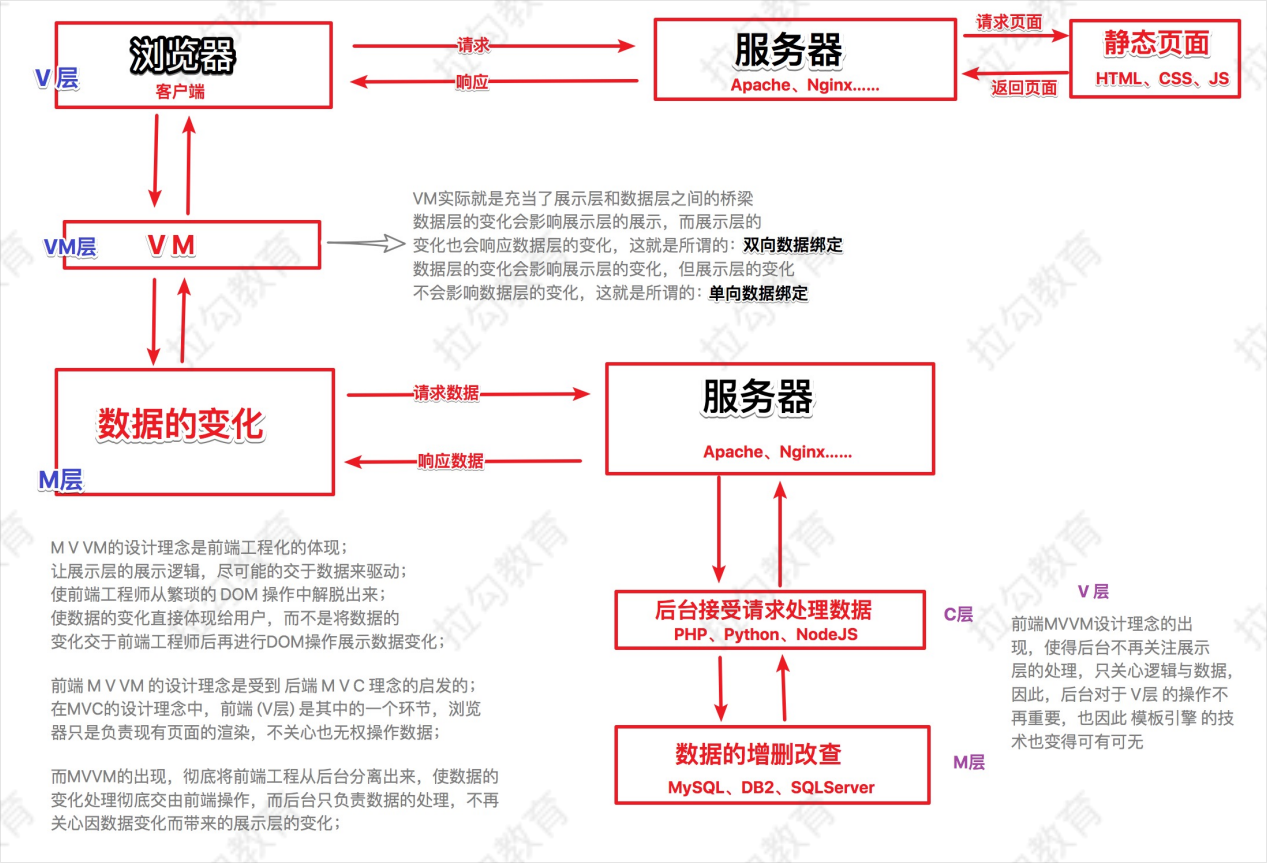
传统的服务端 MVC 架构模型：View

```
models  数据模型，专门提供数据支持的
controllers 控制器模块，处理不同的页面请求的或者处理接口请求
views 视图文件
```



> 核心理念：单一职责，分工协作
>
> 优点：
>
> - 更好的开发效率
> - 更好的可维护性

MVVM 模式常见于用于构建用户界面的客户端应用。



MVVM 模型：

字面意义是这样的：

MVVM 是 Model-View-ViewModel 的缩写，MVVM 是一种设计思想。

- Model 层代表数据模式，也可以在 Model 中定义数据修改和操作的业务逻辑
- View 代表 UI 组件，它负责将数据模型转化为 UI 展现出来
- ViewModel 是一个同步 View 和 Model 的对象。

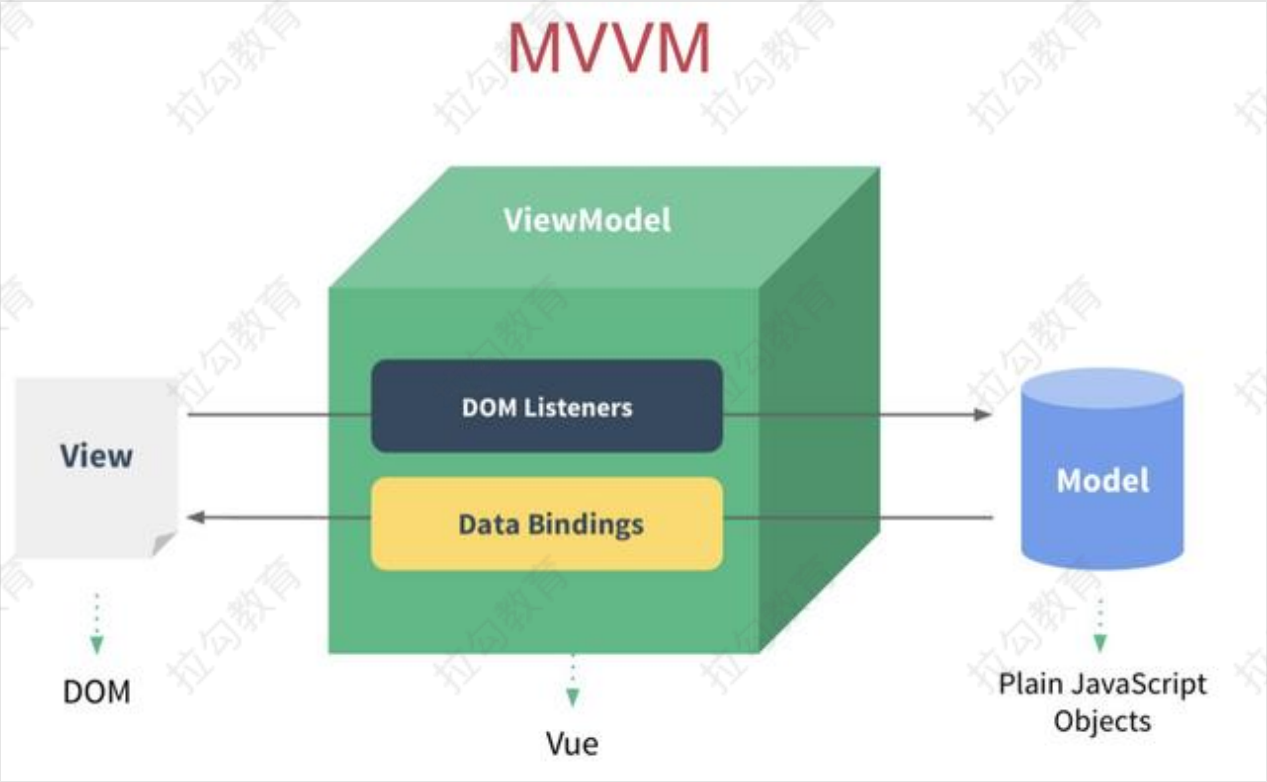
我的理解：

- 前端开发早期的时候都是**操作 DOM**
- 后来使用 jQuery 让我们**提高了操作 DOM 的效率**，但从开发角度**还是在大量的手动操作 DOM**
- MVVM 模式让以往手动操作 DOM 的方式彻底解脱了，它**不要用户自己操作 DOM**，而是**将普通数据绑定到 ViewModel 上，会自动将数据渲染到页面中，视图变化会通知 ViewModel 层更新数据，数据变化也会通过 ViewModel 层更新视图**，因此 MVVM 中最重要的角色就是 ViewModel，真正意义上把视图和数据实现了解耦，提高了开发效率。
- MVVM 模式主要用于构建用户界面的前端应用
  - 微软的 WPF，构建客户端应用的
  - 手机应用，iOS APP、Android App
  - Web 应用
  - ...

核心：

- MVVM 模式让我们从繁琐的 DOM 操作中彻底解放了
- MVVM 也叫数据驱动视图

下面是在 Vue 中的 MVVM：



用 Vue 中组件代码来表示 MVVM 的话就是这样的：

```
View 视图层
<template>
  <div>
    <h1>{{ message }}</h1>
    <ul>
      <li v-for="item in list" :key="item.id">{{ item.title }}</li>
    </ul>
  </div>
</template>

<script>
// Model 层
// ViewModel 层，就是 Vue 本身
export default {
  data () {
    // 普通 JavaScript 数据都会被转化到 ViewModel 层
    return {
      message: 'hello',
      list: []
    }
  },
```

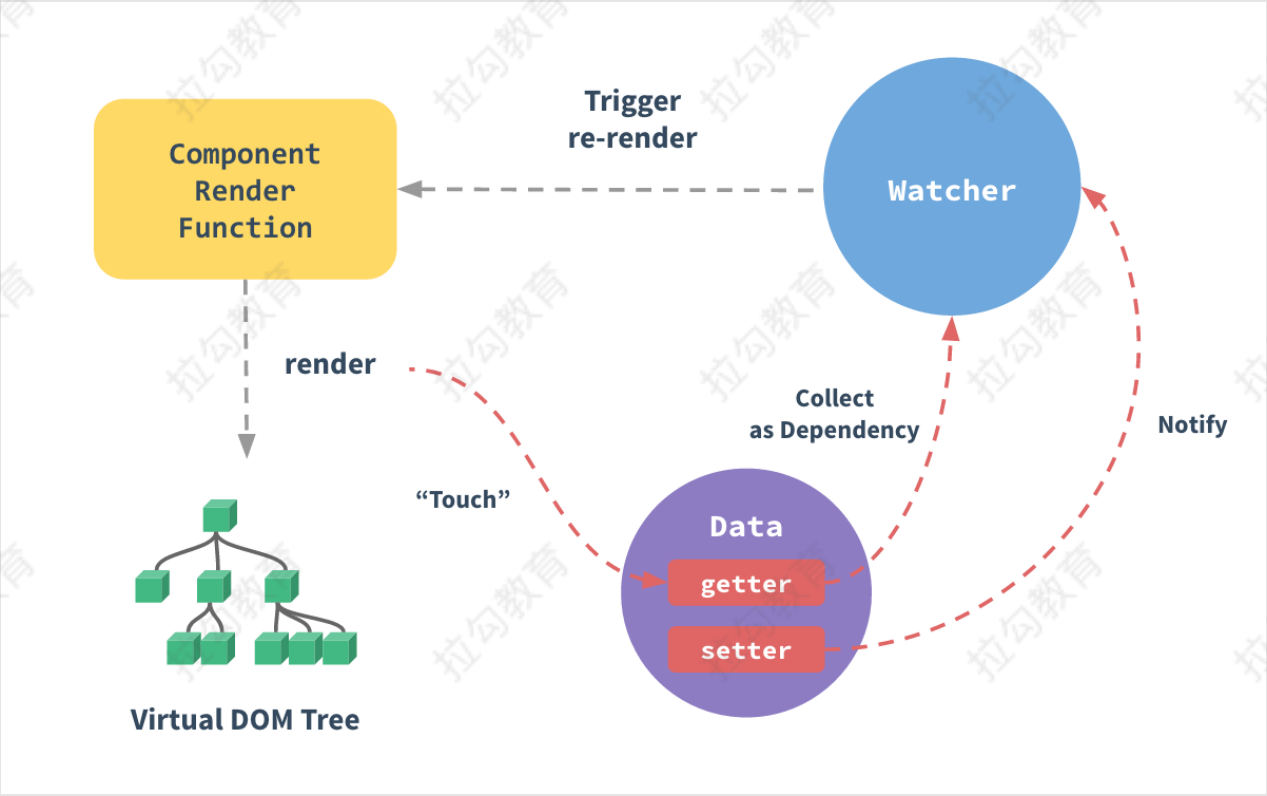
```
  async created () {
    const { data } = await ajax({
      method: 'GET',
      url: 'xxx'
    })
    this.list = data
  }
}
</script>

<style>

</style>
```

5. 谈一谈你对 Vue.js 的响应式数据的理解

核心知识点：

- 在 Vue 2.x 版本中使用的是 Object.defineProperty
- 在 Vue 3.x 版本中使用的是 ECMAScript 6 中新增的 Proxy

先来看一下 Vue.js 官网的解释：https://cn.vuejs.org/v2/guide/reactivity.html。



总结一下：

- 当你把一个普通的 JavaScript 对象传入 Vue 实例作为 `data` 选项，Vue 将遍历此对象所有的 property，并使用 `Object.defineProperty` 把这些 property 全部转为 getter/setter。
  - `Object.defineProperty` 是 ES5 中一个无法 shim 的特性，这也就是 Vue 不支持 IE8 以及更低版本浏览器的原因。
- 当使用这些数据属性时，会进行依赖收集（收集到当前组件的 watcher）
  - 每个组件都对应一个 watcher 实例，它会在组件渲染的过程中把“接触”过的数据记录为依赖
- 之后当依赖项的 setter 触发时，会通知 watcher，从而使它关联的组件重新渲染

下面我们可以通过源码来理解它的原理实现。

核心方法	作用	源码
function Vue () {}	Vue 构造函数	core/instance/index.js:8
Vue.prototype._init	初始化实例成员	core/instance/init.js:16
vm.initState	初始化 props、data、watch、computed	core/instance/state.js:50
vm.initData	初始化用户传入的 data 数据	core/instance/state.js:112
observe		
new Observer	将数据进行监视	core/observer/index.js:124
this.walk	处理对象的监视	core/observer/index.js:55
defineReactive	循环对象属性定义响应式变化	core/observer/index.js:135
Object.defineProperty	核心方法，监视数据（拦截数据的访问和修改）	core/observer/index.js:157
get	收集依赖	
set	通知更新	

简而言之就是两件事儿：

- 拦截属性的获取进行依赖收集
- 拦截属性的修改对相关依赖进行通知更新

6. 浏览器和 Node.js 中的事件循环

JavaScript 单线程：同一时间只能干一件事儿，假如有一个死循环，那你整个程序就崩溃了。

所以 JavaScript 不适合做大量运算的任务。

JavaScript 中有很多的异步任务：发请求、读写文件，这些都是比较耗时的，需要进行 IO 工作。

主线程执行普通的代码，遇到异步任务就把它扔到事件队列中。完了才提取队列中的任务开始执行，异步任务执行多线程（环境提供的，我们也不到）。

异步成功，把 callback 放到队列中。

中间过程，主线程会不断的取提取队列中的任务，如果是回调，就直接执行。

7. Vue 中如何实现监测数组变化

如果被问到 Vue 怎么实现数据双向绑定，大家肯定都会回答 通过 Object.defineProperty() 对数据进行劫持，但是 Object.defineProperty() 只能对属性进行数据劫持，不能对整个对象进行劫持，同理无法对数组进行劫持，但是我们在使用 Vue 框架中都知道，Vue 能检测到对象和数组（部分方法的操作）的变化，那它是怎么实现的呢？我们查看相关代码如下：

```
/**
 * Observe a list of Array items.
 */
observeArray (items: Array<any>) {
  for (let i = 0, l = items.length; i < l; i++) {
    observe(items[i])  // observe 功能为监测数据的变化
  }
}

/**
 * 对属性进行递归遍历
 */
let childOb = !shallow && observe(val) // observe 功能为监测数据的变化
```

通过以上 Vue 源码部分查看，我们就能知道 Vue 框架是通过遍历数组 和递归遍历对象，从而达到利用 Object.defineProperty() 也能对对象和数组（部分方法的操作）进行监听。

Vue 将被侦听的数组的变更方法进行了包裹，所以它们也将会触发视图更新。这些被包裹过的方法包括：

- `push()`
- `pop()`
- `shift()`
- `unshift()`
- `splice()`
- `sort()`
- `reverse()`

总结一下：

- 使用函数劫持的方式，重写了数组的方法
- Vue 将 data 中的数组，进行了原型链重写。指向了自己定义的数组原型方法，这样当调用数组 API 的时候，可以通知依赖更新。如果数组中包含着引用类型。会对数组中的引用类型再次进行监

控拦截处理

8. Proxy 与 Object.defineProperty 优劣对比

Proxy 的优势如下:

- Proxy 可以直接监听对象而非属性；
- Proxy 可以直接监听数组的变化；
- 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；
- 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；
- 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；

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。

9. Vue 怎么用 vm.$set() 解决对象新增属性不能响应的问题？

受现代 JavaScript 的限制 ，Vue **无法检测到对象属性的添加或删除**。由于 Vue 会在初始化实例时对属性执行 getter/setter 转化，所以属性必须在 data 对象上存在才能让 Vue 将它转换为响应式的。但是 Vue 提供了 `Vue.set (object, propertyName, value) / vm.$set (object, propertyName, value)` 来实现为对象添加响应式属性，那框架本身是如何实现的呢?

我们查看对应的 Vue 源码：`vue/src/core/instance/observer/index.js`

```
export function set (target: Array<any> | Object, key: any, val: any): any {
  // target 为数组
  if (Array.isArray(target) && isValidArrayIndex(key)) {
    // 修改数组的长度, 避免索引>数组长度导致splcie()执行有误
    target.length = Math.max(target.length, key)
    // 利用数组的splice变异方法触发响应式
    target.splice(key, 1, val)
    return val
  }
  // key 已经存在，直接修改属性值
  if (key in target && !(key in Object.prototype)) {
    target[key] = val
    return val
  }
  const ob = (target: any).__ob__
  // target 本身就不是响应式数据, 直接赋值
```

```
  if (!ob) {
    target[key] = val
    return val
  }
  // 对属性进行响应式处理
  defineReactive(ob.value, key, val)
  ob.dep.notify()
  return val
}
```

我们阅读以上源码可知，vm.$set 的实现原理是：

- 如果目标是数组，直接使用数组的 splice 方法触发相应式；
- 如果目标是对象，会先判读属性是否存在、对象是否是响应式，最终如果要对属性进行响应式处理，则是通过调用 defineReactive 方法进行响应式处理（ defineReactive 方法就是 Vue 在初始化对象时，给对象属性采用 Object.defineProperty 动态添加 getter 和 setter 的功能所调用的方法）

10. Vue 中 nextTick 的实现原理

在下次 DOM 更新循环结束之后执行延迟回调。在修改数据之后立即使用这个方法，获取更新后的 DOM。

```
// 修改数据
vm.msg = 'Hello'
// DOM 还没有更新
Vue.nextTick(function () {
  // DOM 更新了
})

// 作为一个 Promise 使用 (2.1.0 起新增，详见接下来的提示)
Vue.nextTick()
  .then(function () {
    // DOM 更新了
  })
```

> 2.1.0 起新增：如果没有提供回调且在支持 Promise 的环境中，则返回一个 Promise。请注意 Vue 不自带 Promise 的 polyfill，所以如果你的目标浏览器不原生支持 Promise (IE：你们都看我干嘛)，你得自己提供 polyfill。

可能你还没有注意到，Vue 在更新 DOM 时是**异步**执行的。只要侦听到数据变化，Vue 将开启一个队列，并缓冲在同一事件循环中发生的所有数据变更。如果同一个 watcher 被多次触发，只会被推入到队列中一次。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。

例如，当你设置 `vm.someData = 'new value'`，该组件不会立即重新渲染。当刷新队列时，组件会在下一个事件循环“tick”中更新。多数情况我们不需要关心这个过程，但是如果你想基于更新后的 DOM 状态来做点什么，这就可能会有些棘手。虽然 Vue.js 通常鼓励开发人员使用“数据驱动”的方式思考，避免直接接触 DOM，但是有时我们必须要这么做。为了在数据变化之后等待 Vue 完成更新 DOM，可以在数据变化之后立即使用 `Vue.nextTick(callback)`。这样回调函数将在 DOM 更新完成后被调用。例如：

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

```
var vm = new Vue({
  el: '#example',
  data: {
    message: '123'
  }
})
vm.message = 'new message' // 更改数据
vm.$el.textContent === 'new message' // false
Vue.nextTick(function () {
  vm.$el.textContent === 'new message' // true
})
```

在组件内使用 `vm.$nextTick()` 实例方法特别方便，因为它不需要全局 `Vue`，并且回调函数中的 `this` 将自动绑定到当前的 Vue 实例上：

```
Vue.component('example', {
  template: '<span>{{ message }}</span>',
  data: function () {
    return {
      message: '未更新'
    }
  },
  methods: {
    updateMessage: function () {
      this.message = '已更新'
      console.log(this.$el.textContent) // => '未更新'
      this.$nextTick(function () {
        console.log(this.$el.textContent) // => '已更新'
      })
    }
  }
})
```

因为 `$nextTick()` 返回一个 `Promise` 对象，所以你可以使用新的 ES2017 async/await 语法完成相同的事情：

```
methods: {
  updateMessage: async function () {
    this.message = '已更新'
    console.log(this.$el.textContent) // => '未更新'
    await this.$nextTick()
    console.log(this.$el.textContent) // => '已更新'
  }
}
```

总结：

nextTick 方法主要是使用了宏任务和微任务，定义了一个异步方法，多次调用 nextTick 方法会将方法存入队列中，通过这个异步方法清空当前队列。所以这个 nextTick 方法就是异步方法。

院系解析：

- core/util/next-tick.js:87
  - nextTick调用它传入 cb
  - callback.push 将回调函数存入数组中
  - TimeFunc() 调用 timeFunc()
    - Promise 回调
    - MutationObserve
    - setImmediate
    - setTimeout
    - 最终执行 nextTick 中传入的方法
  - 返回 Promise
    - 回调也可以
    - 也可以 Promise

11. `v-if` 和 `v-show` 的区别

`v-if` 是“真正”的条件渲染，因为它会确保在切换过程中条件块内的事件监听器和子组件适当地被销毁和重建。

- true：渲染
- false：销毁不渲染，元素就不存在了

`v-if` 也是**惰性的**：如果在初始渲染时条件为假，则什么也不做——直到条件第一次变为真时，才会开始渲染条件块。

相比之下，`v-show` 就简单得多——不管初始条件是什么，元素总是会被渲染，并且只是简单地基于 CSS 进行切换。

- true：display: block
- false: display: none

一般来说，`v-if` 有更高的切换开销，而 `v-show` 有更高的初始渲染开销。因此，如果需要非常频繁地切换，则使用 `v-show` 较好；如果在运行时条件很少改变，则使用 `v-if` 较好。

12. Vue 中的 key 有什么作用

key 是为 Vue 中 vnode 的唯一标记，通过这个 key，我们的 diff 操作可以更准确、更快速。Vue 的 diff 过程可以概括为：oldCh 和 newCh 各有两个头尾的变量 oldStartIndex、oldEndIndex 和 newStartIndex、newEndIndex，它们会新节点和旧节点会进行两两对比，即一共有4种比较方式：newStartIndex 和oldStartIndex 、newEndIndex 和 oldEndIndex 、newStartIndex 和 oldEndIndex 、newEndIndex 和 oldStartIndex，如果以上 4 种比较都没匹配，如果设置了key，就会用 key 再进行比较，在比较的过程中，遍历会往中间靠，一旦 StartIdx > EndIdx 表明 oldCh 和 newCh 至少有一个已经遍历完了，就会结束比较。具体有无 key 的 diff 过程，可以查看作者写的另一篇详解虚拟 DOM 的文章《深入剖析：Vue核心之虚拟DOM》

所以 Vue 中 key 的作用是：key 是为 Vue 中 vnode 的唯一标记，通过这个 key，我们的 diff 操作可以更准确、更快速

更准确：因为带 key 就不是就地复用了，在 sameNode 函数 `a.key === b.key` 对比中可以避免就地复用的情况。所以会更加准确。

更快速：利用 key 的唯一性生成 map 对象来获取对应节点，比遍历方式更快，源码如下：

```
function createKeyToOldIdx (children, beginIdx, endIdx) {
  let i, key
  const map = {}
  for (i = beginIdx; i <= endIdx; ++i) {
    key = children[i].key
    if (isDef(key)) map[key] = i
  }
  return map
}
```

13. 怎样理解 Vue 的单向数据流?

所有的 prop 都使得其父子 prop 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改变父级组件的状态，从而导致你的应用的数据流向难以理解。

额外的，每次父级组件发生更新时，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 会在浏览器的控制台中发出警告。子组件想修改时，只能通过 $emit 派发一个自定义事件，父组件接收到后，由父组件修改。

有两种常见的试图改变一个 prop 的情形：

- **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 属性并将这个 prop 用作其初始值：

```
props: ['initialCounter'],
data: function () {
  return {
    counter: this.initialCounter
  }
}
复制代码
```

- **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

```
props: ['size'],
computed: {
  normalizedSize: function () {
    return this.size.trim().toLowerCase()
  }
}
```

14. 谈一谈你对 Vue 生命周期的理解

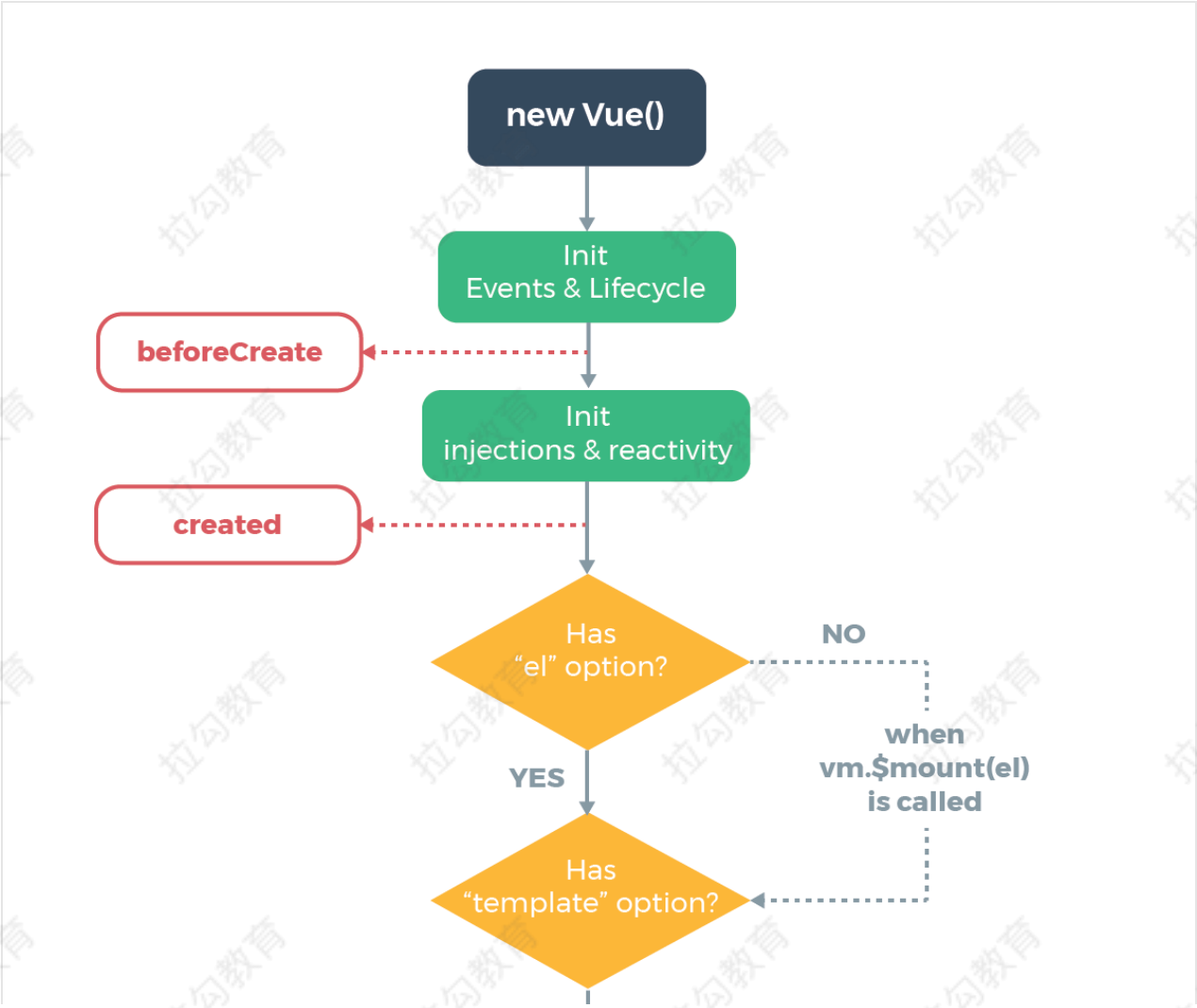
（1）生命周期是什么?

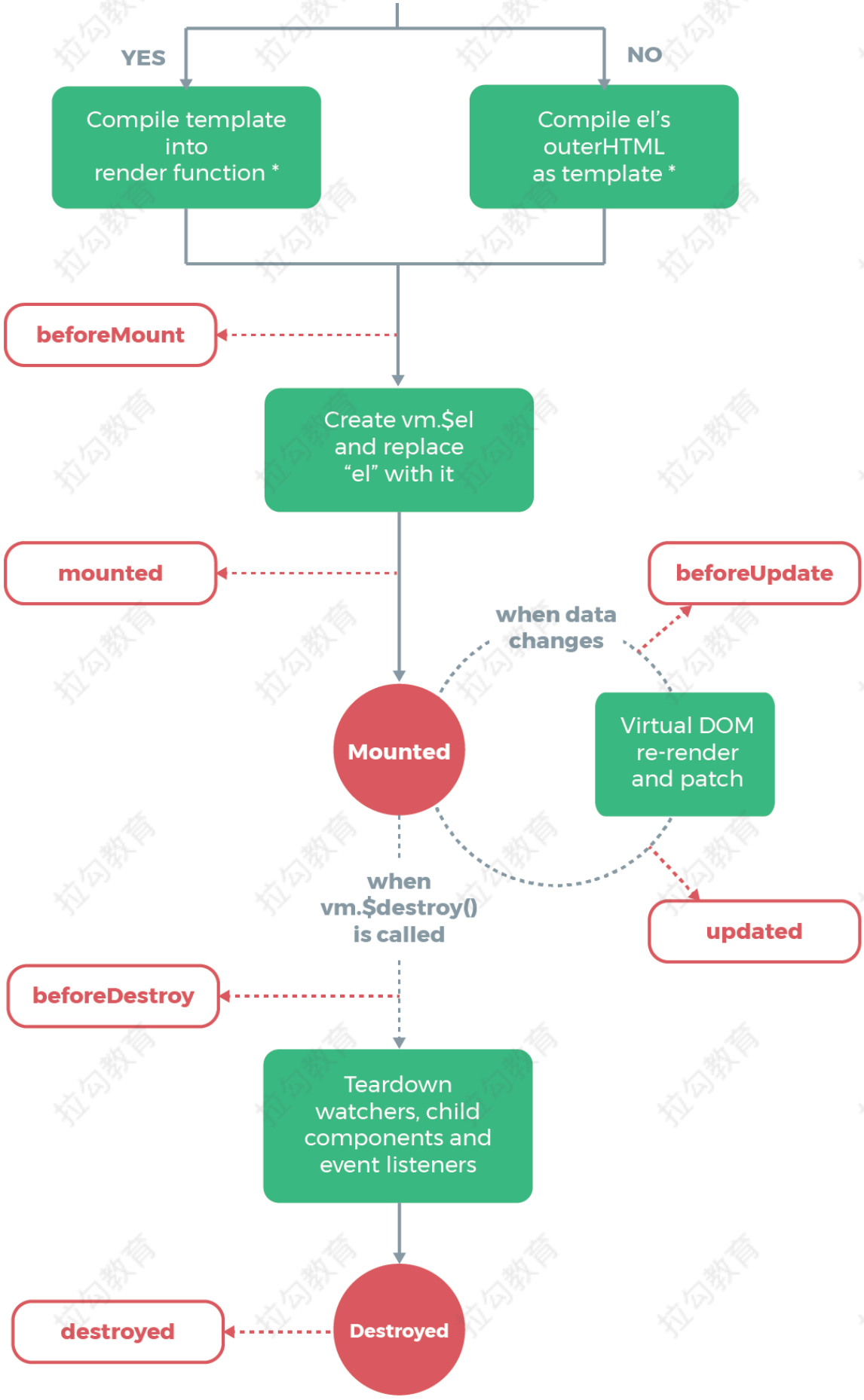
Vue 实例有一个完整的生命周期，也就是从开始创建、初始化数据、编译模版、挂载 Dom -> 渲染、更新 -> 渲染、卸载等一系列过程，我们称这是 Vue 的生命周期。

（2）各个生命周期的作用

生命周期	描述
beforeCreate	组件实例被创建之初，组件的属性生效之前
created	组件实例已经完全创建，属性也绑定，但真实 dom 还没有生成，$el 还不可用
beforeMount	在挂载开始之前被调用：相关的 render 函数首次被调用
mounted	el 被新创建的 vm.$el 替换，并挂载到实例上去之后调用该钩子
beforeUpdate	组件数据更新之前调用，发生在虚拟 DOM 打补丁之前
update	组件数据更新之后
activited	keep-alive 专属，组件被激活时调用
deactivated	keep-alive 专属，组件被销毁时调用
beforeDestory	组件销毁前调用
destoryed	组件销毁后调用

3）生命周期示意图




YES
NO
Compile template into render function *
Compile el's outerHTML as template *
beforeMount
Create vm.$el and replace "el" with it
mounted
beforeUpdate
when data changes
Mounted
Virtual DOM re-render and patch
when vm.$destroy() is called
updated
beforeDestroy
Teardown watchers, child components and event listeners
destroyed
Destroyed

```
* template compilation is performed ahead-of-time if using
a build step, e.g. single-file components
```

15. 什么时候使用 beforeDestroy

- 可能在当前页面中使用了 $on 方法，那需要在组件销毁前解绑
- 清除自己定义的定时器
- 解除事件绑定：scroll、mousemove

16. 组件的 data 为什么是函数

防止组件重用的时候导致数据相互影响。

```
export default {
  data: {
    a: 1,
    b: 2
  }
}

组件a：data
组件b：data

export default {
  data () {
    return {
      a: 1,
      b: 2
    }
  }
}

组件a：data()
组件b：data()

function data () {
  return {
    a: 1,
    b: 2
  }
}

const data1 = data()
const data2 = data()

data1 === data2
```

17. Vue 中事件绑定原理

```
vm.$on('事件名称', 处理函数)

vm.$emit('事件名称', 可选参数)
```

18. 为什么 Vue 采用异步渲染

因为如何不采用异步渲染，那么每次更新数据都会对当前组件进行重新渲染，所以为了性能考虑，Vue 会在本轮数据更新后，再去异步更新视图。

Vue 是组件级更新，当组件中的数据变化就会更新组件。

```
this.a = 1
this.b = 2

// 数据修改之后立即操作 DOM，拿到的是旧的
```

组件中的数据绑定的都是同一个 watcher，Vue 可以把相同的 watcher 更新过滤掉。所以这是为了提高性能的一个考虑。

19. computed 和 watch 的区别

- computed 是属性
  - 当你需要根据已有数据产生一些派生数据的时候， 可以使用计算属性
  - 注意：计算属性不支持异步操作，因为计算属性一般要绑定到模板中
  - 更重要的一点是：计算属性会缓存调用的结果，提高性能
  - 计算属性都必须有返回值，没有返回值就没有意义
- watch 是一个功能
  - watch 不需要返回值，根据某个数据变化执行 xxx 逻辑
  - watch 可以执行异步操作

20. 为什么不推荐同时使用 v-if 和 v-for

当 `v-if` 与 `v-for` 一起使用时， `v-for` 具有比 `v-if` 更高的优先级。请查阅列表渲染指南以获取详细信息。

```
<div v-for="xxx" v-if="xxx"></div>

<div v-for="xxx">
  // 控制内部元素是否渲染
  <元素 v-if="xxx"></元素>
</div>

// 控制整个循环块是否渲染
<元素 v-if="xx">
  <元素 v-for="xxx"></元素>
</元素>
```

21. 关于虚拟 DOM

21.1. 是什么

虚拟 dom 是相对于浏览器所渲染出来的真实 dom 的，在react，vue等技术出现之前，我们要改变页面展示的内容只能通过遍历查询 dom 树的方式找到需要修改的 dom 然后修改样式行为或者结构，来达到更新 ui 的目的。

这种方式相当消耗计算资源，因为每次查询 dom 几乎都需要遍历整颗 dom 树，如果建立一个与 dom 树对应的虚拟 dom 对象（ js 对象），以对象嵌套的方式来表示 dom 树，那么每次 dom 的更改就变成了 js 对象的属性的更改，这样一来就能查找 js 对象的属性变化要比查询 dom 树的性能开销小。

为什么操作 DOM 性能开销大?

其实并不是查询 dom 树性能开销大而是 dom 树的实现模块和 js 模块是分开的这些跨模块的通讯增加了成本，以及 dom 操作引起的浏览器的回流和重绘，使得性能开销巨大，原本在 pc 端是没有性能问题的，因为 pc 的计算能力强，但是随着移动端的发展，越来越多的网页在智能手机上运行，而手机的性能参差不齐，会有性能问题。

新技术如何解决性能问题?

angular，react，vue 等框架的出现就是为了解决这个问题的。

他们的思想是每次更新 dom 都尽量避免刷新整个页面，而是有针对性的去刷新那被更改的一部分，来释放掉被无效渲染占用的 gpu，cup 性能。

- angular
  - angular 采用的机制是 脏值检测查机制 所有使用了 ng 指令的 scope data 和 {{}} 语法的 scope data 都会被加入脏检测的队列
- vue
  - vue 采用的是虚拟 dom 通过重写 setter， getter 实现观察者监听 data 属性的变化生成新的虚拟 dom 通过 h 函数创建真实 dom 替换掉dom树上对应的旧 dom。
- react
  - react 也是通过虚拟 dom 和 setState 更改 data 生成新的虚拟 dom 以及 diff 算法来计算和生成需要替换的 dom 做到局部更新的。

下面是一个关于虚拟 DOM 的例子：

这个虚拟 dom 是我自己实现的不是 vue 或者 react 的内部实现。

```
const HelloWorld = {
    nodeName:'div',
    attrs:{
        className:'',
    },
    css:{
        width: '100px',
        height: '40px',
        color: 'green'
    },
    events:{
        onclick:()=>{ console.log('Hello virtual DOM') }
    },
    childrens:[
        {
            nodeName:'text',
            attrs:{
                innerText:'HelloWorld',
            },
        }
    ]
}
复制代码
```

下面我们来尝试实现一个解析虚拟 dom 的 render 函数。

如何解析这个虚拟 DOM 呢?

```
function render(vNode) {
    // 创建dom
    const dom = document.createElement(vNode.nodeName)
    const { attrs, css, events, childrens } = vNode

    // 添加属性
    for(const attrName in attrs){
        dom[attrName] = attrs[attrName]
    }

    // 添加行内样式
    for(const attrName in css){
        dom.style[attrName] = css[attrName]
    }

    // 添加事件
    for(const eventName in events){
```

```
            dom[eventName] = events[eventName]
        }

        if(childrens){
            for(const children of childrens) {
                // 生成子节点
                const childrenNode = render(children)
                // 绑定子节点
                dom.append(childrenNode)
            }
        }

        return dom
}
```

大家可以将这段代码复制到浏览器测试一下。

```
const dom = render(HelloWorld)
document.body.append(dom)
```

21.2. 优缺点

优点：

- **保证性能下限：** 框架的虚拟 DOM 需要适配任何上层 API 可能产生的操作，它的一些 DOM 操作的实现必须是普适的，所以它的性能并不是最优的；但是比起粗暴的 DOM 操作性能要好很多，因此框架的虚拟 DOM 至少可以保证在你不需要手动优化的情况下，依然可以提供还不错的性能，即保证性能的下限；
- **无需手动操作 DOM：** 我们不再需要手动去操作 DOM，只需要写好 View-Model 的代码逻辑，框架会根据虚拟 DOM 和 数据双向绑定，帮我们以可预期的方式更新视图，极大提高我们的开发效率；
- **跨平台：** 虚拟 DOM 本质上是 JavaScript 对象,而 DOM 与平台强相关，相比之下虚拟 DOM 可以进行更方便地跨平台操作，例如服务器渲染、weex 开发等等。

缺点:

- **无法进行极致优化：** 虽然虚拟 DOM + 合理的优化，足以应对绝大部分应用的性能需求，但在一些性能要求极高的应用中虚拟 DOM 无法进行针对性的极致优化。

21.3. 实现原理

虚拟 DOM 的实现原理主要包括以下 3 部分：

- 用 JavaScript 对象模拟真实 DOM 树，对真实 DOM 进行抽象；
- diff 算法 — 比较两棵虚拟 DOM 树的差异；
- pach 算法 — 将两个虚拟 DOM 对象的差异应用到真正的 DOM 树。

如果对以上 3 个部分还不是很了解的同学，可以查看本文作者写的另一篇详解虚拟 DOM 的文章《深入剖析：Vue核心之虚拟DOM》

22. 你对 Vue 项目做过哪些优化

如果没有对 Vue 项目没有进行过优化总结的同学，可以参考本文作者的另一篇文章《 Vue 项目性能优化 — 实践指南 》，文章主要介绍从 3 个大方面，22 个小方面详细讲解如何进行 Vue 项目的优化。

（1）代码层面的优化

- v-if 和 v-show 区分使用场景
- computed 和 watch 区分使用场景
- v-for 遍历必须为 item 添加 key，且避免同时使用 v-if
- 长列表性能优化
- 事件的销毁
- 图片资源懒加载
- 路由懒加载
- 第三方插件的按需引入
- 优化无限列表性能
- 服务端渲染 SSR or 预渲染

（2）Webpack 层面的优化

- Webpack 对图片进行压缩
- 减少 ES6 转为 ES5 的冗余代码
- 提取公共代码
- 模板预编译
- 提取组件的 CSS
- 优化 SourceMap
- 构建结果输出分析
- Vue 项目的编译优化

（3）基础的 Web 技术的优化

- 开启 gzip 压缩
- 浏览器缓存
- CDN 的使用
- 使用 Chrome Performance 查找性能瓶颈

23. 对于即将到来的 vue3.0 特性你有什么了解的吗?

Vue 3.0 正走在发布的路上，Vue 3.0 的目标是让 Vue 核心变得更小、更快、更强大，因此 Vue 3.0 增加以下这些新特性：

（1）监测机制的改变

3.0 将带来基于代理 Proxy 的 observer 实现，提供全语言覆盖的反应性跟踪。这消除了 Vue 2 当中基于 Object.defineProperty 的实现所存在的很多限制：

- 只能监测属性，不能监测对象
- 检测属性的添加和删除；
- 检测数组索引和长度的变更；
- 支持 Map、Set、WeakMap 和 WeakSet。

新的 observer 还提供了以下特性：

- 用于创建 observable 的公开 API。这为中小规模场景提供了简单轻量级的跨组件状态管理解决方案。
- 默认采用惰性观察。在 2.x 中，不管反应式数据有多大，都会在启动时被观察到。如果你的数据集很大，这可能会在应用启动时带来明显的开销。在 3.x 中，只观察用于渲染应用程序最初可见部分的数据。
- 更精确的变更通知。在 2.x 中，通过 Vue.set 强制添加新属性将导致依赖于该对象的 watcher 收到变更通知。在 3.x 中，只有依赖于特定属性的 watcher 才会收到通知。
- 不可变的 observable：我们可以创建值的“不可变”版本（即使是嵌套属性），除非系统在内部暂时将其“解禁”。这个机制可用于冻结 prop 传递或 Vuex 状态树以外的变化。
- 更好的调试功能：我们可以使用新的 renderTracked 和 renderTriggered 钩子精确地跟踪组件在什么时候以及为什么重新渲染。

（2）模板

模板方面没有大的变更，只改了作用域插槽，2.x 的机制导致作用域插槽变了，父组件会重新渲染，而 3.0 把作用域插槽改成了函数的方式，这样只会影响子组件的重新渲染，提升了渲染的性能。

同时，对于 render 函数的方面，vue3.0 也会进行一系列更改来方便习惯直接使用 api 来生成 vdom 。

（3）对象式的组件声明方式

vue2.x 中的组件是通过声明的方式传入一系列 option，和 TypeScript 的结合需要通过一些装饰器的方式来做，虽然能实现功能，但是比较麻烦。3.0 修改了组件的声明方式，改成了类式的写法，这样使得和 TypeScript 的结合变得很容易。

此外，vue 的源码也改用了 TypeScript 来写。其实当代码的功能复杂之后，必须有一个静态类型系统来做一些辅助管理。现在 vue3.0 也全面改用 TypeScript 来重写了，更是使得对外暴露的 api 更容易结合 TypeScript。静态类型系统对于复杂代码的维护确实很有必要。

（4）其它方面的更改

vue3.0 的改变是全面的，上面只涉及到主要的 3 个方面，还有一些其他的更改：

- 支持自定义渲染器，从而使得 weex 可以通过自定义渲染器的方式来扩展，而不是直接 fork 源码来改的方式。
- 支持 Fragment（多个根节点）和 Protal（在 dom 其他部分渲染组建内容）组件，针对一些特殊

的场景做了处理。

- 基于 treeshaking 优化，提供了更多的内置功能。