

# Vue 面试通杀秘籍

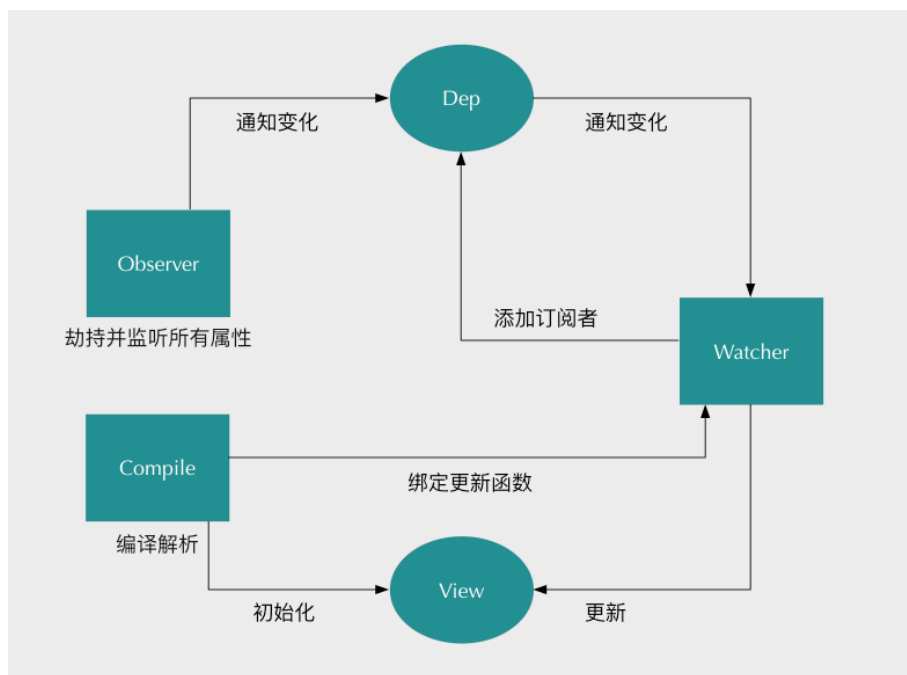
## 理论篇：

### 1. 说说对 Vue 渐进式框架的理解（腾讯医典）

- a) 渐进式的含义：主张最少，没有多做职责之外的事
- b) Vue 有些方面是不如 React，不如 Angular.但它是渐进的，没有强主张，你可以在原有系统的上面，把一两个组件改用它实现，当 jQuery 用；
- c) 也可以整个用它全家桶开发，当 Angular 用；还可以用它的视图，搭配你自己设计的整个下层用。
- d) 你可以在底层数据逻辑的地方用 OO 和设计模式的那套理念，也可以函数式，都可以，它只是个轻量视图而已，只做了自己该做的事，没有做不该做的事，仅此而已。

### 2. vue 的双向绑定的原理（腾讯医典）

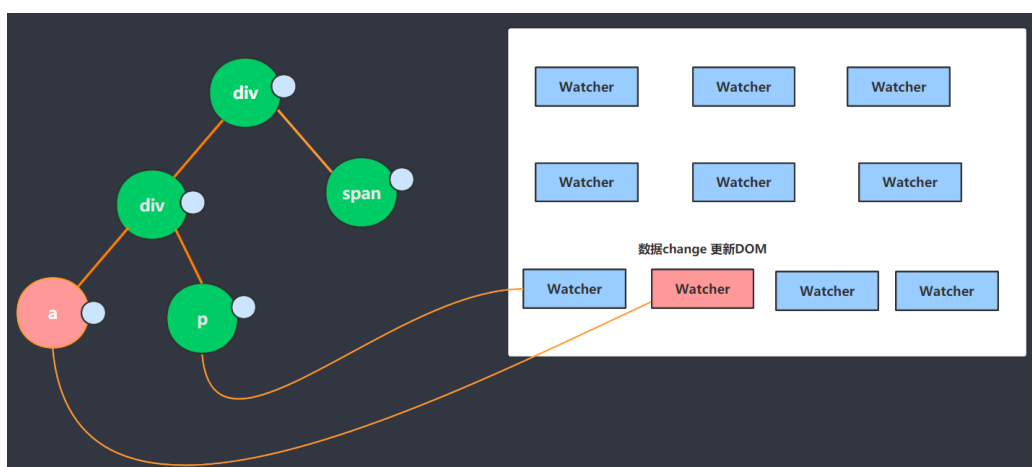
- a) 数据双向绑定是通过数据劫持结合发布者-订阅者模式的方式来实现的。
- b) 具体实现流程：
  - i. 实现一个监听器 Observer，用来劫持并监听所有属性，如果有变动的，就通知订阅者
  - ii. 实现一个订阅者 Watcher，可以收到属性的变化通知并执行相应的函数，从而更新视图
  - iii. 实现一个解析器 Compile，可以扫描和解析每个节点的相关指令，并根据初始化模板数据以及初始化相应的订阅器
- c) 把下面的流程图说清楚就差不多了：



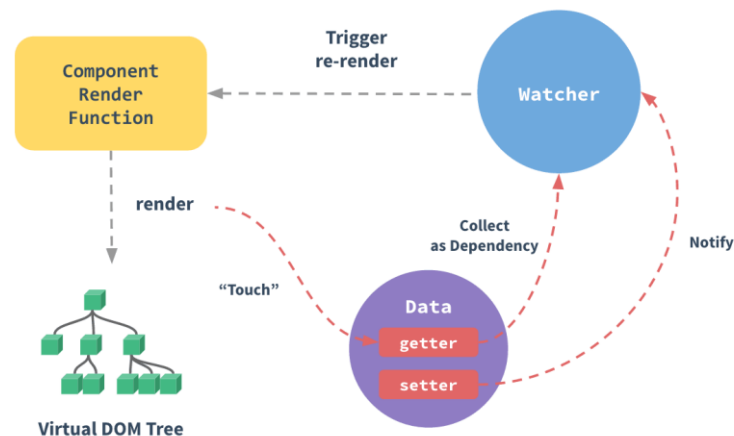
### 3. Vue1.x 的设计理念是什么？2.x 的呢？（58 同城）

#### Vue1.x 设计理念

- 早期 Vue 是 采用数据绑定、依赖收集的方式去观察数据变化并保留对实际 DOM 元素的引用，当有数据变化时进行对应的操作。
- 少量数据更新对比 Virtual DOM 性能更好，坏处是大量数据更新、初始渲染性能、对比 Virtual DOM 性能更差
- 图解：



## Vue2.x 设计理念



1. Vue2.0 引入了虚拟 dom
  - 通过建立虚拟 dom 树, `document.createDocumentFragment()`, 方法创建虚拟 dom 树。一旦被监测的数据改变, 会通过 `Object.defineProperty` 定义的数据拦截, 截取到数据的变化。
2. 截取到的数据变化, 从而通过订阅——发布者模式, 触发 Watcher (观察者), 从而改变虚拟 dom 的中的具体数据。
3. 最后, 通过更新虚拟 dom 的元素值, 从而改变最后渲染 dom 树的值, 完成双向绑定。

## 应用篇:

### 1. Key 的作用是什么? 可以用数组的 index (下标) 代替么? (美团)

- a) key 的作用主要是为了高效的更新虚拟 DOM。另外 vue 中在使用相同标签名元素的过渡切换时, 也会使用到 key 属性, 其目的也是为了让 vue 可以区分它们。否则 vue 只会替换其内部属性而不会触发过渡效果
- b) key 不能用 index 代替, index 在同一个页面会有重复的情况, 违背了高效渲染的初衷。

## 2. Vue 组件中 data 为什么必须是函数？（58 同城）

- a) 在 `new Vue()`，`data` 是可以作为一个对象进行操作的，然而在 component 中，`data` 只能以函数的形式存在，不能直接将对象赋值给它。
- b) 当 `data` 选项是一个函数的时候，每个实例可以维护一份被返回对象的独立的拷贝，这样各个实例中的 `data` 不会相互影响，是独立的。

## 3. \$route 和 \$router 的区别是什么？（深信服）

- a) `$router` 为 `VueRouter` 的实例，是一个全局路由对象，包含了路由跳转的方法、钩子函数等。
- b) `$route` 是路由信息对象||跳转的路由对象，每一个路由都会有一个 `route` 对象，是一个局部对象，包含 `path`, `params`, `hash`, `query`, `fullPath`, `matched`, `name` 等路由信息参数。

# Vue 3.0

## 1. 为什么要用 Proxy API 替代 defineProperty API? (响应式优化)

- a) defineProperty 的局限性的最大原因是它只能针对单例属性做监听, Vue2.x 中对 data 中的属性做了遍历 + 递归, 为每个属性设置了 getter、setter。这也就是为什么 Vue 只能对 data 中预定义过的属性做出响应的原因。
- b) Proxy 的监听是针对一个对象的, 那么对这个对象的所有操作会进入监听操作, 这就完全可以代理所有属性, 将会带来很大的性能提升和更优的代码。

## 2. Vue3.0 编译做了哪些优化?

### a) 生成 block tree

- i. Vue.js 2.x 的数据更新并触发重新渲染的粒度是组件级的, 单个组件内部需要遍历该组件的整个 vnode 树。
- ii. Vue.js 3.0 做到了通过编译阶段对静态模板的分析, 编译生成了 Block tree。Block tree 是一个将模版基于动态节点指令切割的嵌套区块, 每个区块内部的节点结构是固定的。每个区块只需要追踪自身包含的动态节点。

### 2.1 传统 Virtual DOM 的性能瓶颈

```
<template>
  <div id="content">
    <p class="text">Lorem ipsum</p>
    <p class="text">Lorem ipsum</p>
    <p class="text">{{ message }}</p>
    <p class="text">Lorem ipsum</p>
    <p class="text">Lorem ipsum</p>
  </div>
</template>
```

- Diff <div>
  - Diff props of <div>
  - Diff children of <div>
    - Diff <p>
      - Diff props of <p>
      - Diff children of <p>
    - Repeat n times...

- 传统 vdom 的性能跟模版大小正相关, 跟动态节点的数量无关。在一些组件整个模版内只有少量动态节点的情况下, 这些遍历都是性能的浪费

### 2.1.1 根本原因

```
function render() {
  const children = []
  for (let i = 0; i < 5; i++) {
    children.push(h('p', {
      class: 'text'
    }, i === 2 ? this.message : 'Lorum ipsum'))
  }
  return h('div', { id: 'content' }, children)
}
```

- 根本原因: JSX 和手写的 render function 是完全动态的, 过度的灵活性导致运行时可以用于优化的信息不足

### 2.1.2 优化 (动静结合)

```
<template>
  <div id="content">
    <p class="text">Lorem ipsum</p>
    <p class="text">Lorem ipsum</p>
    <p class="text">{{ message }}</p>
    <p class="text">Lorem ipsum</p>
    <p class="text">Lorem ipsum</p>
  </div>
</template>
```

- Diff <p> textContent

- 节点结构完全不会改变
- 只有一个动态节点

#### b) slot 编译优化

- i. Vue.js 2.x 中, 如果有一个组件传入了 slot, 那么每次父组件更新的时候, 会强制使子组件 update, 造成性能的浪费。
- ii. Vue.js 3.0 优化了 slot 的生成, 使得非动态 slot 中属性的更新只会触发子组件的更新。动态 slot 指的是在 slot 上面使用 v-if, v-for, 动态 slot 名字等会导致 slot 产生运行时动态变化但是又无法被子组件 track 的操作。