vue、react、angular三者有什么区别？

vue.js通过简单的API（应用程序编程接口）提供高效的数据绑定和灵活的组件系统。

0、Vuejs特点

Vue.js的特性如下：

1.轻量级的框架

2.双向数据绑定

3.指令系统

4.组件化

5.客户端路由

6.状态管理

1、与AngularJS的区别

特点：

1.良好的应用程序结构

2.双向数据绑定

3.指令

4.HTML模板

5.可嵌入、注入和测试

相同点：

1.都支持指令：内置指令和自定义指令。

2.都支持过滤器：内置过滤器和自定义过滤器。

3.都支持双向数据绑定。

4.都不支持低端浏览器。

不同点：

1.AngularJS的学习成本高，比如增加了Dependency Injection特性，而Vue.js本身提供的API都比较简单、直观。

2.在性能上，AngularJS依赖对数据做脏检查，所以Watcher越多越慢。

Vue.js使用基于依赖追踪的观察并且使用异步队列更新。所有的数据都是独立触发的。

对于庞大的应用来说，这个优化差异还是比较明显的。

2、与React的区别

相同点：

1.React采用特殊的JSX语法，Vue.js在组件开发中也推崇编写.vue特殊文件格式，对文件内容都有一些约定，两者都需要编译后使用。

2.中心思想相同：一切都是组件，组件实例之间可以嵌套。

3.都提供合理的钩子函数，可以让开发者定制化地去处理需求。

4.都不内置列数AJAX，Route等功能到核心包，而是以插件的方式加载。

5.在组件开发中都支持mixins的特性。

不同点：

React依赖Virtual DOM,而Vue.js使用的是DOM模板。React采用的Virtual DOM会对渲染出来的结果做脏检查。

Vue.js在模板中提供了指令，过滤器等，可以非常方便，快捷地操作DOM。

vue 的认识与介绍

1、渐进式框架Vue

Vue.js 是一套构建用户界面的渐进式框架。与其他重量级框架不同的是，Vue 采用自底向上增量开发的设计。

渐进式 -- 阶梯式向前，vue是轻量级的，它有很多独立的功能或库，能根据不同的需求使用不同的功能

渐进式表现：声明式渲染——组件系统——客户端路由———-大数据状态管理———-构建工具

2、vue中两个核心点

1）响应式数据绑定

当数据发生变化是，vue自动更新视图

原理：

Vue2.0:利用了 Object.definedProperty 中的setter/getter 代理数据，监控对数据的操作。

Vue3.0:直接采用了Proxy来监听数据的变化

2）组合的视图组件

ui页面映射为组件书

划分组件可维护、可重用、可测试

3、虚拟DOM

大量的Dom操作会导致页面加载十分缓慢，通常在数据变化的时候，JS会通过处理DOM来更新视图数据，但是在更新页面后会导致页面重新渲染，这样的结果就是无需修改数据的地方也会重新渲染DOM节点。这样性能方面就会很受影响。

虚拟DOM--利用在内存中生成与真实DOM与之对应的数据结构，这个在内存中生成的结构称之为虚拟DOM。

当数据发生变化时，能够智能的计算出重新渲染组件的最小代价并应用到DOM操作上（相关知识点 diff算法）

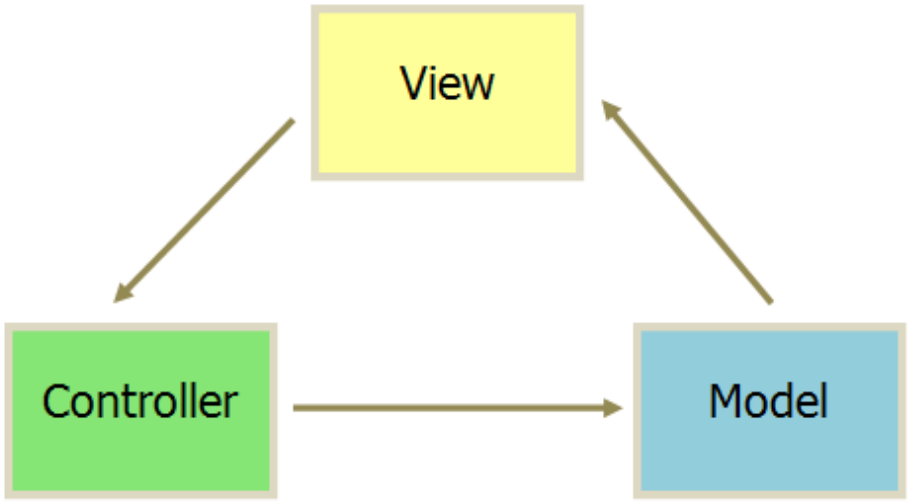
4、MVVM MVC MVP 之前的区别

1）MVC

MVC模式的意思是，软件可以分成三个部分。

- 视图（View）：用户界面。
- 控制器（Controller）：业务逻辑
- 模型（Model）：数据保存

通信方式：



1. View 传送指令到 Controller
2. Controller 完成业务逻辑后，要求 Model 改变状态
3. Model 将新的数据发送到 View，用户得到反馈

所有通信都是单向的

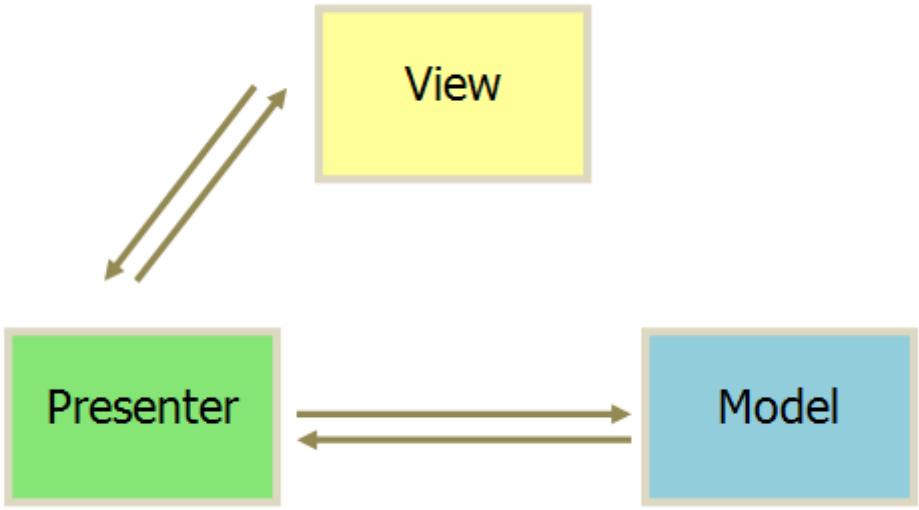
互动模式：

（1）通过 View 接受指令，传递给 Controller。

（2）直接通过controller接受指令。

2）MVP

MVP 模式将 Controller 改名为 Presenter，同时改变了通信方向。



- 各部分之间的通信，都是双向的。
- View 与 Model 不发生联系，都通过 Presenter 传递。
- View 非常薄，不部署任何业务逻辑，称为"被动视图"（Passive View），即没有任何主动性，而 Presenter非常厚，所有逻辑都部署在那里。

3）MVVM

MVVM 模式将 Presenter 改名为 ViewModel，基本上与 MVP 模式完全一致。

- Model：模型层，负责处理业务逻辑以及和服务器端进行交互--服务器请求
- View：视图层，负责将数据模型转化为UI展示出来，可以简单的理解为HTML页面 --视图
- ViewModel：视图模型层，用来连接Model和View，是Model和View之间的通信桥梁 -- JS、数据，对数据进行二次封装

唯一的区别是，它采用双向绑定（data-binding）：View的变动，自动反映在 ViewModel，反之亦然

Vue 采用的就是这种模式

5、vue是如何实现响应式数据的呢？（响应式数据原理）

VUE 2.0

通过`Object.defineProperty`重新定义data中所有的属性，简单来说就是通过遍历所有data的数据，并设置其setting和geting属性来实现监听数据的变化

但是这种方式具有极大的局限性，由于 JavaScript 的限制，Vue **不能检测**数组和对象的变化。

- 对于对象 无法监听对象的删除和添加
- 对于数组 利用索引直接设置一个数组项以及修改数组的长度时无法检测到数组的变化

不止如此，Object.defineProperty通过遍历所有的属性，当数据量一大起来，加载速度就无比的慢。

还有，由于Object.defineProperty是监听所有属性的变化，那么如果数据量巨大的话，所占的内存也会无比的多

VUE 3.0

Proxy 对象用于创建一个对象的代理，从而实现基本操作的拦截和自定义（如属性查找、赋值、枚举、函数调用等）。

也就是无论访问对象的什么属性，之前定义的或是新增的属性，都会走到拦截中进行处理。这就解决了之前所无法监听的问题

详细的对比:[vue响应式原理v2.0与v3.0的区别]

6、Vue的事件绑定原理

`Vue`中通过`v-on`或其语法糖`@`指令来给元素绑定事件并且提供了事件修饰符，基本流程是进行模板编译生成`AST(语法书)`，生成`render`函数后并执行得到`VNode`，`VNode`生成真实`DOM`节点或者组件时候使用`addEventListener`方法进行事件绑定。

额外的知识：

修饰符

> - `.stop`: 调用`event.stopPropagation()`，即阻止事件冒泡。
> - `.prevent`: 调用`event.preventDefault()`，即阻止默认事件。
> - `.capture`: 添加事件侦听器时使用`capture`模式，即使用事件捕获模式处理事件。
> - `.self`: 只当事件是从侦听器绑定的元素本身触发时才触发回调。
> - `.{keyCode | keyAlias}`: 只当事件是从特定键触发时才触发回调。
> - `.native`: 监听组件根元素的原生事件，即注册组件根元素的原生事件而不是组件自定义事件的。
> - `.once`: 只触发一次回调。
> - `.left(2.2.0)`: 只当点击鼠标左键时触发。
> - `.right(2.2.0)`: 只当点击鼠标右键时触发。
> - `.middle(2.2.0)`: 只当点击鼠标中键时触发。
> - `.passive(2.3.0)`: 以`{ passive: true }`模式添加侦听器，表示`listener`永远不会调用`preventDefault()`。

7、v-model中的实现原理及如何自定义v-model

v-model原理

```
<input v-model="sth" />
//  等同于
<input :value="sth" @input="sth = $event.target.value" />
```

也就是说， v-model="sth" 是 :value="sth" @input="sth = $event.target.value" 的缩写。

`v-model` 在内部为不同的输入元素使用不同的属性并抛出不同的事件：

- text 和 textarea 元素使用 `value` 属性和 `input` 事件；
- checkbox 和 radio 使用 `checked` 属性和 `change` 事件；
- select 字段将 `value` 作为 prop 并将 `change` 作为事件

实现原理

- v-model只不过是一个语法糖而已,真正的实现靠的还是
    - v-bind:绑定响应式数据
    - 触发oninput 事件并传递数据

自定义组件实现v-model

```
<my-component v-model="price"></my-component>
//  拆解如下
<my-component :value="price" @input="price = $event.target.value"></my-
component>

//  根据这个我们可以在子组件中，进行拼凑value属性，input方法。
Vue.component('my-component', {
  template: `
    <span>
      <input
        type="text"
        :value="value"
        @input="$emit('input', $event.target.value)"
      >
    </span>
  `,
  props: ['value'],
})
```

8、为什么Vue采用异步渲染呢

根本原因：为了性能

`Vue` 是组件级更新，如果不采用异步更新，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，`Vue` 会在本轮数据更新后，在异步更新视图

流程：

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，若多个属性依赖一个`watcher`，则如果队列中没有该`watcher`就会将该`watcher`添加到队列中，然后便会在`$nextTick`方法的执行队列中加入一个`flushSchedulerQueue`方法(这个方法将会触发在缓冲队列的所有回调的执行)，然后将`$nextTick`方法的回调加入`$nextTick`方法中维护的执行队列，`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，更新完成后会调用`updated`钩子。

数据更新 -->调用`notify`,遍历`watcher` --> `uplate`调用`queueWatcher`将`watcher`方静队列中 -->`watcher`去重 --> `$nextTick`执行队列中添加`flushSchedulerQueue`-->触发`beforeUpdate` -->`watcher.run()` 开始执行`watcher` -->渲染完成-->调用`update`

9、了解nextTick吗

假如此时我们有一个需求，需要在页面渲染完成后取得页面的`DOM`元素，而由于渲染是异步的，我们不能直接在定义的方法中同步取得这个值的，于是就有了`vm.$nextTick`方法，`Vue`中`$nextTick`方法将回调延迟到下次`DOM`更新循环之后执行，也就是在下次`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，能够获取更新后的`DOM`。

简单来说就是当数据更新时，在`DOM`中渲染完成后，执行回调函数。 通过一个简单的例子来演示`$nextTick`方法的作用，首先需要知道`Vue`在更新`DOM`时是异步执行的，也就是说在更新数据时其不会阻塞代码的执行，直到执行栈中代码执行结束之后，才开始执行异步任务队列的代码，所以在数据更新时，组件不会立即渲染，此时在获取到`DOM`结构后取得的值依然是旧的值，而在`$nextTick`方法中设定的回调函数会在组件渲染完成之后执行，取得`DOM`结构后取得的值便是新的值。

```
<body>
    <div id="app"></div>
</body>


<script type="text/javascript">
    var vm = new Vue({
        el: '#app',
        data: {
            msg: 'Vue'
        },
        template:`
            <div>
                <div ref="msgElement">{{msg}}</div>
                <button @click="updateMsg">updateMsg</button>
            </div>
        `,
        methods:{
            updateMsg: function(){
                this.msg = "Update";
                console.log("DOM未更新: ", this.$refs.msgElement.innerHTML)  //
Vue
                this.$nextTick(() => {
                    console.log("DOM已更新: ", this.$refs.msgElement.innerHTML)
//Update
                })
            }
        },

    })
</script>
```

这里涉及到了事件循环（`Event Loop`）的知识点

- 执行栈就是在主线程执行同步任务的数据结构，函数调用形成了一个由若干帧组成的栈。
- 后台线程就是浏览器实现对于`setTimeout`、`setInterval`、`XMLHttpRequest`等等的执行线程。
- 宏队列，一些异步任务的回调会依次进入宏队列，等待后续被调用，包括`setTimeout`、`setInterval`、`setImmediate(Node)`、`requestAnimationFrame`、`UI rendering`、`I/O`等操作。
- 微队列，另一些`异步任务的回调(重点：是异步任务的回调，并不是异步任务会进入微队列)`会依次进入微队列，等待后续调用，包括`Promise`、`process.nextTick(Node)`、`Object.observe`、`MutationObserver`等操作。

当`Js`执行时，进行如下流程:

1. 首先将执行栈中代码同步执行，将这些代码中异步任务加入后台线程中。
2. 执行栈中的同步代码执行完毕后，执行栈清空，并开始扫描微队列。
3. 取出微队列队首任务，放入执行栈中执行，此时微队列是进行了出队操作。

4. 当执行栈执行完成后，继续出队微队列任务并执行，直到微队列任务全部执行完毕。
5. 最后一个微队列任务出队并进入执行栈后微队列中任务为空，当执行栈任务完成后，开始扫面微队列为空，继续扫描宏队列任务，宏队列出队，放入执行栈中执行，执行完毕后继续扫描微队列为空则扫描宏队列，出队执行。
6. 不断往复`...`。

了解异步任务的执行队列后，回到中`$nextTick`方法，当用户数据更新时，`Vue`将会维护一个缓冲队列，对于所有的更新数据将要进行的组件渲染与`DOM`操作进行一定的策略处理后加入缓冲队列，然后便会在`$nextTick`方法的执行队列中加入一个`flushSchedulerQueue`方法(这个方法将会触发在缓冲队列的所有回调的执行)，然后将`$nextTick`方法的回调加入`$nextTick`方法中维护的执行队列，在异步挂载的执行队列触发时就会首先会首先执行`flushSchedulerQueue`方法来处理`DOM`渲染的任务，然后再去执行`$nextTick`方法构建的任务，这样就可以实现在`$nextTick`方法中取得已渲染完成的`DOM`结构。

知识点：事件循环

10、Vue的生命周期

什么时候被调用

- beforeCreate ：实例初始化之后，数据观测之前调用
- created：实例创建完之后调用。实例完成：数据观测、属性和方法的运算、`watch/event`事件回调。无`$el`.
- beforeMount：挂在开始之前被调用，相关的render函数首次被调用（虚拟DOM），实例已完成以下的配置： 编译模板，把data里面的数据和模板生成html，完成了el和data 初始化，注意此时还没有挂在html到页面上
- mounted：挂在完成，也就是模板中的HTML渲染到HTML页面中，此时一般可以做一些ajax操作，mounted只会执行一次。。
- beforeUpdate：数据更新前调用，发生在虚拟DOM重新渲染和打补丁，在这之后会调用改钩子。
- updated：由于数据更改导致的虚拟DOM重新渲染和打补丁，在这之后会调用改钩子。
- beforeDestroy：实例销毁前调用，实例仍然可用。
- destroyed：实例销毁之后调用，调用后，Vue实例指示的所有东西都会解绑，所有事件监听器和所有子实例都会被移除

每个生命周期内部可以做什么？

- created：实例已经创建完成，因为他是最早触发的，所以可以进行一些数据、资源的请求。
- mounted：实例已经挂载完成，可以进行一些DOM操作。
- beforeUpdate：可以在这个钩子中进一步的更改状态，不会触发重渲染。
- updated：可以执行依赖于DOM的操作，但是要避免更改状态，可能会导致更新无线循环。
- destroyed：可以执行一些优化操作，清空计时器，解除绑定事件。

ajax放在哪个生命周期？

一般放在`mounted`中，保证逻辑统一性，因为生命周期是同步执行的，`ajax`是异步执行的。单数服务端渲染`ssr`同一放在`created`中，因为服务端渲染不支持`mounted`方法。

什么时候使用beforeDestroy？

当前页面使用`$on`，需要解绑事件。清楚定时器。解除事件绑定，`scroll mousemove`。

11、父子组件生命周期调用顺序

渲染顺序：先父后子，完成顺序：先子后父

```
父beforeCreate->父created->父beforeMount->子beforeCreate->子created->子beforeMount->子mounted->父mounted
```

更新顺序：父更新导致子更新，子更新完成后父

```
父beforeUpdate->子beforeUpdate->子updated->父updated
```

销毁顺序：先父后子，完成顺序：先子后父

```
父beforeDestroy->子beforeDestroy->子destroyed->父destroyed
```

12、Vue组件通信

1.prop/$emit

父组件：通过`prop`的方式向子组件传递数据

子组件：通过`$emit`可以向父组件通信。

优点：

传值取值方便简洁明了

缺点：

1. 由于数据是单向传递，如果子组件需要改变父组件的props值每次需要给子组件绑定对应的监听事件。
2. 如果父组件需要给孙组件传值，需要子组件进行转发，较为不便。

2.$parent/$children

在子组件中通过`$parent`调用了父组件的函数，并在父组件通过`$children`获取子组件实例的数组

3.provide/inject

父组件中通过provide来提供变量, 然后再子组件中通过inject来注入变量。这里inject注入的变量不像`$attrs`，只能向下一层；inject不论子组件嵌套有多深，都能获取到。

4.$attrs/$listeners

跨一级传输数据

5.vuex

vuex实现了单向的数据流，在全局定义了一个State对象用来存储数据，当组件要修改State中的数据时，必须通过Mutation进行操作。

6.$refs

用在子组件上，获取的就是组件的实例对象。获取组件实例，调用组件的属性、方法

7.Event Bus

跨组件通信`Event Bus`（Vue.prototype.bus = new Vue）其实基于*bus=newVu**e*）其实基于on与$emit

常见使用场景分为以下三类:

- 父子组件通信: props; $parent/$children; provide/inject; $ref; $attrs/$listeners
- 兄弟组件通信: EventBus; Vuex
- 跨级通信: EventBus; Vuex; provide/inject; $attrs/$listeners

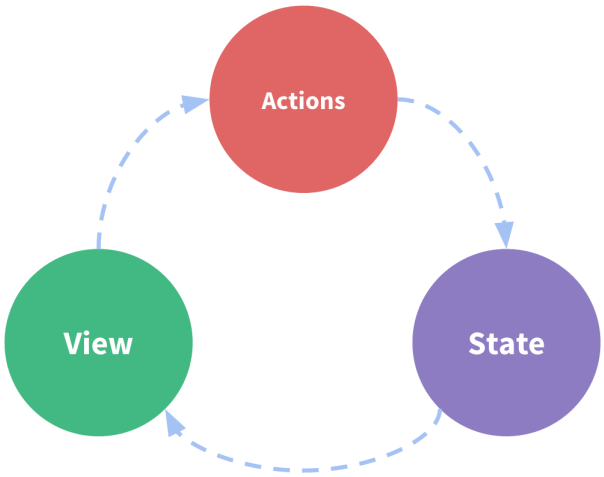
详细：vue组件通信

13.Vuex 工作原理

Vuex 是一个专为 Vue.js 应用程序开发的**状态管理模式。**

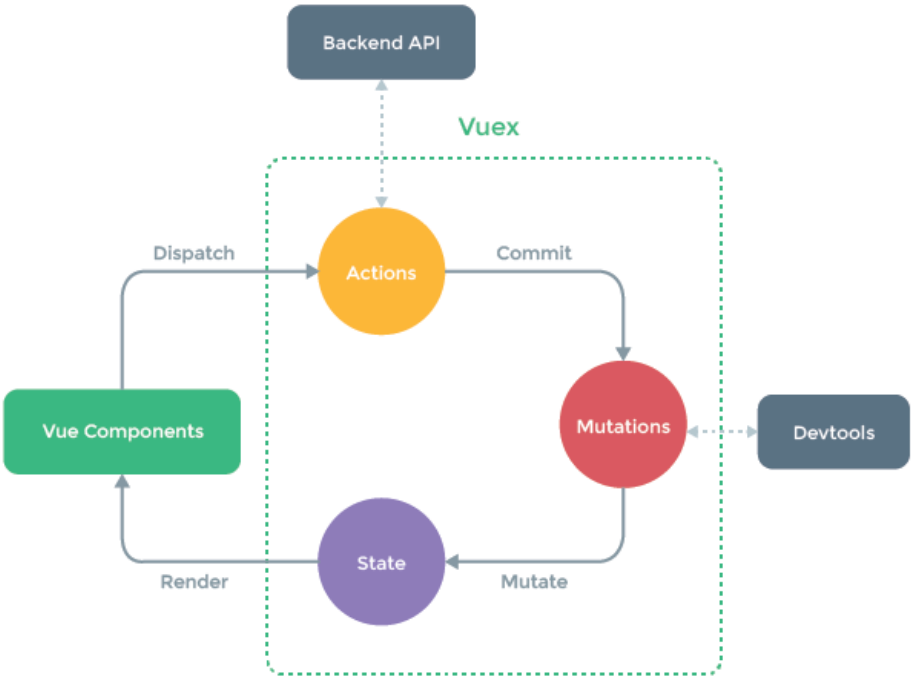
状态自管理应用包含以下几个部分：

- 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。下图单向数据流示意图：



vuex，多组件共享状态，因-单向数据流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。



14.虚拟DOM的理解

实际上它只是一层对真实`DOM`的抽象，以`JavaScript`对象(`VNode`节点)作为基础的树，用对象的属性来描述节点，最终可以通过一系列操作使这棵树映射到真实环境上

在`Javascript`对象中，虚拟`DOM`表现为一个`Object`对象。并且最少包含标签名(`tag`)、属性(`attrs`)和子元素对象(`children`)三个属性，不同框架对这三个属性的名命可能会有差别

例子：

定义真实`DOM`

```
<div id="app">
    <p class="p">节点内容</p>
    <h3>{{ foo }}</h3>
</div>
```

实例化vue

```
const app = new Vue({
    el:"#app",
    data:{
        foo:"foo"
    }
})
```

观察`render`的`render`，得到虚拟`DOM`

```
(function anonymous(
) {
    with(this){return _c('div',{attrs:{"id":"app"}},[_c('p',{staticClass:"p"},
                    [_v("节点内容")]),_v(" "),_c('h3',[_v(_s(foo))])])}})
```

通过 `VNode`，`vue` 可以对这颗抽象树进行创建节点,删除节点以及修改节点的操作， 经过 `diff` 算法得出一些需要修改的最小单位,再更新视图，减少了 `dom` 操作，提高了性能

15.diff算法

`diff` 算法是一种通过同层的树节点进行比较的高效算法

特点：

- 比较只会在同层级进行, 不会跨层级比较
- 在diff比较的过程中，循环从两边向中间比较

`diff` 算法的在很多场景下都有应用，在 `vue` 中，作用于虚拟 `dom` 渲染成真实 `dom` 的新旧 `VNode` 节点比较

比较方式：

深度优先，同层比较

1. 比较只会在同层级进行, 不会跨层级比较
2. 比较的过程中，循环从两边向中间收拢

比较实例：[你了解vue的diff算法吗]

流程：

- 当数据发生改变时，订阅者 `watcher` 就会调用 `patch` 给真实的 `DOM` 打补丁
- 通过 `isSameVnode` 进行判断，相同则调用 `patchVnode` 方法
- `patchVnode`

  做了以下操作：

  - 找到对应的真实 `dom`，称为 `el`
  - 如果都有都有文本节点且不相等，将 `el` 文本节点设置为 `Vnode` 的文本节点
  - 如果 `oldVnode` 有子节点而 `VNode` 没有，则删除 `el` 子节点
  - 如果 `oldVnode` 没有子节点而 `VNode` 有，则将 `VNode` 的子节点真实化后添加到 `el`
  - 如果两者都有子节点，则执行 `updateChildren` 函数比较子节点
- `updateChildren`

  主要做了以下操作：

  - 设置新旧 `VNode` 的头尾指针
  - 新旧头尾指针进行比较，循环向中间靠拢，根据情况调用 `patchVnode` 进行 `patch` 重复流程、调用 `createElem` 创建一个新节点，从哈希表寻找 `key` 一致的 `VNode` 节点再分情况操作