六星教育 WEB 前端面试圣经

Vue 基础

Vue.js 路由的钩子函数

首先可以控制导航跳转， beforeEach ， afterEach 等，一般用于页面 title 的修改。一些需要登录才能调整页面的重定向功能。

beforeEach 主要有 3 个参数 to ， from ， next 。

to ： route 即将进入的目标路由对象。

from ： route 当前导航正要离开的路由。

next ： function 一定要调用该方法 resolve 这个钩子。执行效果依赖 n ext 方法的调用参数。可以控制网页的跳转

vuex 是什么？怎么使用？哪种功能场景使用它？

只用来读取的状态集中放在 store 中； 改变状态的方式是提交 mutations ，这是个同步的事物； 异步逻辑应该封装在 action 中。

在 main.js 引入 store ，注入。新建了一个目录 store ， … export

场景有：单 应用中，组件之间的状态、音乐播放、登录状态、加入购物

state ： Vuex 使用单一状态树,即每个应用将仅仅包含一个 store 实例，但单一状态

树和模块化并不冲突。存放的数据状态，不可以直接修改里面的数据。

mutations ： mutations 定义的方法动态修改 Vuex 的 store 中的状态或数据

getters ：类似 vue 的计算属性，主要用来过滤一些数据。

action ： actions 可以理解为通过将 mutations 里面处里数据的方法变成可异步的处理数据的方法，简单的说就是异步操作数据。 view 层通过 store.dispath 来分发 action

modules ：项目特别复杂的时候，可以让每一个模块拥有自己的 state 、mutation 、 action 、getters ，使得结构非常清晰，方便管理

Vue.js $route 和 $router 的区别

$route 是“路由信息对象”，包括 path ， params ， hash ， query ，fullPath ， matched ， name 等路由信息参数。

而 $router 是“路由实例”对象包括了路由的跳转方法，钩子函数等

Vue.js 如何让 CSS 只在当前组件中起作用?

将当前组件的 <style> 修改为 <style scoped>

Vue.js keep-alive 的作用是什么?

包裹动态组件时，会缓存不活动的组件实例,主要用于保留 组件状态或避免重新渲染

比如有一个列表和一个详情，那么用户就会经常执行打开详情=>返回列表=> 打开详情...这样的话列表和详情都是一个频率很高的 面，那么就可以对列表 组件使用 `进行缓存，这样用户每次返回列表的 时候，都能从缓存中快速渲染，而不是重新渲染

Vue.js 指令 v-el 的作用是什么?

提供一个在页面上已存在的 DOM 元素作为 Vue 实例的挂载目标.可以是 CSS 选择器，也可以是一个 HTMLElement 实例

在 Vue.js 中使用插件的步骤

采用 ES6 的 import ... from ... 语法或 CommonJS 的 require() 方法引入插件

使用全局方法 Vue.use(plugin) 使用插件，可以传入一个选项对象 Vue.use(MyPlugin, { someOption: true })

Vue.js NextTick 作用

nextTick 可以让我们在下次 DOM 更新循环结束之后执行延迟回调，用于获 得更新后的 DOM

Vue.js 的优点是什么？

低耦合。视图(View)可以独立于 Model 变化和修改，一个 ViewModel 可以绑定到不同的 “View” 上，当 View 变化的时候 Model 可以不变，当 Model 变化的时候 View 也可以不变

可重用性。你可以把一些视图逻辑放在一个 ViewModel 里面，让很多 view 重用这段视图逻辑

可测试。界面素来是比较难于测试的，而现在测试可以针对 ViewModel 来写

Vue.js 路由之间跳转？

声明式（标签跳转）

```
<router-link :to="index">
```

编程式（ js 跳转）

```
router.push('index')
```

Vue.js computed 实现

建立与其他属性（如： data 、 Store ）的联系；

属性改变后，通知计算属性重新计算

实现时，主要如下

初始化 data ， 使用 Object.defineProperty 把这些属性全部转为 getter/setter 。

初始化 computed，遍历 computed 里的每个属性，每个 computed 属性都是一个 watch 实例。每个属性提供的函数作为属性的 getter ，使用Object.defineProperty 转化。

Object.defineProperty getter 依赖收集。用于依赖发生变化时，触发属性重新计 算。

若出现当前 computed 计算属性嵌套其他 computed 计算属性时，先进行其他的依赖收 集

Vue.js complier 实现

模板解析这种事，本质是将数据转化为一段 html ，最开始出现在后端，经过各种处理吐 给前端。随着各种 mv* 的兴起，模板解析交由前端处理。

总的来说， Vue complier 是将 template 转化成一个 render 字符串。

可以简单理解成以下步骤：

parse 过程，将 template 利用正则转化成 AST 抽象语法树。

optimize 过程，标记静态节点，后 diff 过程跳过静态节点，提升性能。

generate 过程，生成 render 字符串

Vue.js 组件中 data 什么时候可以使用对象

组件复用时所有组件实例都会共享 data ，如果 data 是对象的话，就会造成一个组件 修改 data 以后会影响到其他所有组件，所以需要将 data 写成函数，每次用到就调用 一次函数获得新的数据。

当我们使用 new Vue() 的方式的时候，无论我们将 data 设置为对象还是函数都是可 以的，因为 new Vue() 的方式是生成一个根组件，该组件不会复用，也就不存在共享 data 的情况了

深入理解

Vue.js 中是如何检测数组变化?

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 时，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。

```
const arrayProto = Array.prototype
export const arrayMethods = Object.create(arrayProto)
const methodsToPatch = [
'push',
'pop',
'shift',
'unshift',
'splice',
'sort',
'reverse'
]
methodsToPatch.forEach(function (method) { // 重写原型方法
const original = arrayProto[method] // 调用原数组的方法
def(arrayMethods, method, function mutator (...args) {
const result = original.apply(this, args)
const ob = this.__ob__
```

```
        let inserted
        switch (method) {
        case 'push':
        case 'unshift':
        inserted = args
        break
        case 'splice':
        inserted = args.slice(2)
        break
        }
        if (inserted) ob.observeArray(inserted)
        // notify change
        ob.dep.notify() // 当调用数组方法后，手动通知视图更新
        return result
        })
        })

        this.observeArray(value) // 进行深度监控
```

为何 Vue.js 采用异步渲染?

因为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.

所以为了性能考虑。Vue 会在本轮数据更新后，再去异步更新视图!

对应源码

```
update () {
/* istanbul ignore else */
if (this.lazy) {
this.dirty = true
} else if (this.sync) {
this.run()
} else {
queueWatcher(this); // 当数据发生变化时会将 watcher 放到一个队列中批量更新
}
}
export function queueWatcher (watcher: Watcher) {
const id = watcher.id // 会对相同的 watcher 进行过滤
if (has[id] == null) {
has[id] = true
if (!flushing) {
queue.push(watcher)
} else {
let i = queue.length - 1
while (i > index && queue[i].id > watcher.id) {
i--
```

```
        }
        queue.splice(i + 1, 0, watcher)
        }
        // queue the flush
        if (!waiting) {
        waiting = true

        if (process.env.NODE_ENV !== 'production' && !config.async) {
        flushSchedulerQueue()
        return
        }
        nextTick(flushSchedulerQueue) // 调用 nextTick 方法 批量的进行更新
        }
        }
        }
```

Vue.js 中 Computed 的特点

默认 `computed` 也是一个 `watcher` 是具备缓存的，只要当依赖的属性发生变化时才会更新视图

对 应 源 码

```
        function initComputed (vm: Component, computed: Object) {
        const watchers = vm._computedWatchers = Object.create(null)
        const isSSR = isServerRendering()
```

```
for (const key in computed) {
const userDef = computed[key]
const getter = typeof userDef === 'function' ? userDef : userDef.get
if (!isSSR) {
// create internal watcher for the computed property.
watchers[key] = new Watcher(
vm,
getter || noop,
noop,
computedWatcherOptions
)
}

// component-defined computed properties are already defined on the
// component prototype. We only need to define computed properties defined
// at instantiation here.
if (!(key in vm)) {
defineComputed(vm, key, userDef)
} else if (process.env.NODE_ENV !== 'production') {
if (key in vm.$data) {
warn(`The computed property "${key}" is already defined in data.`, vm)
} else if (vm.$options.props && key in vm.$options.props) {
```

```
warn(`The computed property "${key}" is already defined as a prop.`, vm)
}
}
}
}
function createComputedGetter (key) {
return function computedGetter () {
const watcher = this._computedWatchers && this._computedWatchers[key]
if (watcher) {
if (watcher.dirty) { // 如果依赖的值没发生变化,就不会重新求值
watcher.evaluate()
}
if (Dep.target) {
watcher.depend()
}
return watcher.value
}
}
}
```

Vue.js Watch 中的 deep:true 是如何实现的

六星教育 WEB 前端面试圣经

当用户指定了 `watch` 中的 deep 属性为 `true` 时，如果当前监控的值是数组类型。会对对象中的每一项进行求值，此时会将当前 `watcher` 存入到对应属性的依赖中，这样数组中对象发生变化时也会通知数据更新

对应源码

```
get () {
    pushTarget(this) // 先将当前依赖放到 Dep.target 上
    let value
    const vm = this.vm
    try {
      value = this.getter.call(vm, vm)
    } catch (e) {
      if (this.user) {
        handleError(e, vm, `getter for watcher "${this.expression}"`)
      } else {
        throw e
      }
    } finally {
      if (this.deep) { // 如果需要深度监控
        traverse(value) // 会对对象中的每一项取值,取值时会执行对应的 get 方法
      }
      popTarget()
    }
```

```
    return value
}
function _traverse (val: any, seen: SimpleSet) {
  let i, keys
  const isA = Array.isArray(val)
  if ((!isA && !isObject(val)) || Object.isFrozen(val) || val instanceof VNode) {
    return
  }
  if (val.__ob__) {
    const depId = val.__ob__.dep.id
    if (seen.has(depId)) {
      return
    }
    seen.add(depId)
  }
  if (isA) {
    i = val.length
    while (i--) _traverse(val[i], seen)
  } else {
    keys = Object.keys(val)
    i = keys.length
    while (i--) _traverse(val[keys[i]], seen)
```

```
    }
}
```

Vue.js 何时需要使用 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

解除事件的绑定 scroll mousemove ....

Vue.js 中 v-if 和 v-show 的区别

v-if 如果条件不成立不会渲染当前指令所在节点的 dom 元素

v-show 只是切换当前 dom 的显示或者隐藏

v-show 只是在 display: none 和 display: block 之间切换。无论初始条件是什么 都会被渲染出来，后面只需要切换 CSS ， DOM 还是一直保留着的。所以总的来说 v- show 在初始渲染时有更高的开销，但是切换开销很小，更适合于频繁切换的场景。

v-if 的话就得说到 Vue 底层的编译了。当属性初始为 false 时，组件就不会被渲 染，直到条件为 true ，并且切换条件时会触发销毁/挂载组件，所以总的来说在切换时开 销更高，更适合不经常切换的场景。

并且基于 v-if 的这种惰性渲染机制，可以在必要的时候才去渲染组件，减少整个 面的 初始渲染开销。

总结： v-if 按照条件是否渲染， v-show 是 display 的 block 或 none ；

Vue.js 中为什么 v-for 和 v-if 不能连用

v-for 会比 v-if 的优先级高一些,如果连用的话会把 v-if 给每个元素都添加一下,会造成性能问题

对 应 源 码

```
const VueTemplateCompiler = require('vue-template-compiler');
let r1 = VueTemplateCompiler.compile(`<div v-if="false" v-for="i in 3">hello</div>`);
```

```
/**
with(this) {
    return _l((3), function (i) {
        return (false) ? _c('div', [_v("hello")]) : _e()
    })
}
*/
console.log(r1.render);
```

用 vnode 来描述一个 Vue.js DOM 结构

虚拟节点就是用一个对象来描述真实的 dom 元素

对 应 源 码

```
function $createElement(tag,data,...children){
    let key = data.key;
    delete data.key;
    children = children.map(child=>{
        if(typeof child === 'object'){
            return child
        }else{
            return vnode(undefined,undefined,undefined,undefined,child)
        }
    })
    return vnode(tag,props,key,children);
```

```
}
export function vnode(tag,data,key,children,text){
    return {
        tag, // 表示的是当前的标签名
        data, // 表示的是当前标签上的属性
        key, // 唯一表示用户可能传递
        children,
        text
    }
}
```

Vue.js 中 diff 算法的时间复杂度

两个树的完全的 diff 算法是一个时间复杂度为 O(n3),Vue 进行了优化·*O(n3)* *复杂度*的问题转换成 O(n) *复杂度*的问题(只比较同级不考虑跨级问题) 在前端当中， 你很少会跨越层级地移动 Dom 元素。 所以 Virtual Dom 只会对同一个层级的元素进行对比。

Vue.js 中 v-html 会导致哪些问题?

可能会导致 xss 攻击

v-html 会替换掉标签内部的子元素

对 应 源 码

```
let template = require('vue-template-compiler');
let r = template.compile(`<div v-html="'<span>hello</span>'"></div>`)
// with(this){return _c('div',{domProps:{"innerHTML":_s('<span>hello</span>')}})}
console.log(r.render);
```

```
// _c 定义在 core/instance/render.js
// _s 定义在 core/instance/render-helpers/index.js

if (key === 'textContent' || key === 'innerHTML') {
    if (vnode.children) vnode.children.length = 0
    if (cur === oldProps[key]) continue
    // #6601 work around Chrome version <= 55 bug where single textNode
    // replaced by innerHTML/textContent retains its parentNode property
    if (elm.childNodes.length === 1) {
      elm.removeChild(elm.childNodes[0])
    }
}
```

Vue.js 中什么是作用域插槽?

1.插 槽

```
<app><div slot="a">xxxx</div><div slot="b">xxxx</div></app>
slot name="a"
slot name="b"
```

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）

2.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。（插槽的作用域为子组件）

对应源码

1.插槽：

```
const VueTemplateCompiler = require('vue-template-compiler');
let ele = VueTemplateCompiler.compile(`
    <my-component>
        <div slot="header">node</div>
        <div>react</div>
        <div slot="footer">vue</div>
    </my-component>
`)
/**
with(this) {
    return _c('my-component', [_c('div', {
        attrs: {
            "slot": "header"
        },
        slot: "header"
    }, [_v("node")]), _v(" "), _c('div', [_v("react")]), _v(" "), _c('div', {
        attrs: {
            "slot": "footer"
        },
```

```
            slot: "footer"
        }, [_v("vue")])])
    }
    */

    const VueTemplateCompiler = require('vue-template-compiler');
    let ele = VueTemplateCompiler.compile(`
        <div>
            <slot name="header"></slot>
            <slot name="footer"></slot>
            <slot></slot>
        </div>
    `);
    /**
    with(this) {
        return _c('div', [_t("header"), _v(" "), _t("footer"), _v(" "), _t("default")], 2)
    }
    **/
    // _t 定义在  core/instance/render-helpers/index.js
```

作用域插槽：

```
    let ele = VueTemplateCompiler.compile(`
```

```
    <app>
        <div slot-scope="msg" slot="footer">{{msg.a}}</div>
    </app>
`);
/**
with(this) {
    return _c('app', {
        scopedSlots: _u([{ //  作用域插槽的内容会被渲染成一个函数
            key: "footer",
            fn: function (msg) {
                return _c('div', {}, [_v(_s(msg.a))])
            }
        }])
    })
 }
}
*/
const VueTemplateCompiler = require('vue-template-compiler');

VueTemplateCompiler.compile(`
    <div>
        <slot name="footer" a="1" b="2"></slot>
```

```
    </div>
`);
/**
with(this) {
    return _c('div', [_t("footer", null, {
        "a": "1",
        "b": "2"
    })], 2)
}
**/
```

谈谈你对 Vue.js 中 keep-alive 的了解

keep-alive 可以实现组件的缓存，当组件切换时不会对当前组件进行卸载,常用的 2 个属性 include/exclude,2 个生命周期 activated,deactivated

对 应 源 码

core/components/keep-alive.js

```
export default {
  name: 'keep-alive',
  abstract: true, // 抽象组件

  props: {
    include: patternTypes,
```

```
    exclude: patternTypes,
    max: [String, Number]
  },

  created () {
    this.cache = Object.create(null) // 创建缓存列表
    this.keys = [] // 创建缓存组件的 key 列表
  },

  destroyed () { // keep-alive 销毁时 会清空所有的缓存和 key
    for (const key in this.cache) { // 循环销毁
      pruneCacheEntry(this.cache, key, this.keys)
    }
  },

  mounted () { // 会监控 include 和 include 属性 进行组件的缓存处理
    this.$watch('include', val => {
      pruneCache(this, name => matches(val, name))
    })
    this.$watch('exclude', val => {
      pruneCache(this, name => !matches(val, name))
    })
```

```
    },

    render () {
      const slot = this.$slots.default // 会默认拿插槽
      const vnode: VNode = getFirstComponentChild(slot) // 只缓存第一个组件
      const componentOptions: ?VNodeComponentOptions = vnode && vnode.componentOptions
      if (componentOptions) {
        // check pattern
        const name: ?string = getComponentName(componentOptions) // 取出组件的名字
        const { include, exclude } = this
        if ( // 判断是否缓存
          // not included
          (include && (!name || !matches(include, name))) ||
          // excluded
          (exclude && name && matches(exclude, name))
        ) {
          return vnode
        }

        const { cache, keys } = this
        const key: ?string = vnode.key == null
```

```
            // same constructor may get registered as different local components
            // so cid alone is not enough (#3269)
            ? componentOptions.Ctor.cid + (componentOptions.tag ? `::${componentOptions.tag}` : '')
            : vnode.key // 如果组件没 key 就自己通过 组件的标签和 key 和 cid 拼接一个 key

        if (cache[key]) {
            vnode.componentInstance = cache[key].componentInstance // 直接拿到组件实例
            // make current key freshest
            remove(keys, key) // 删除当前的 [b,c,d,e,a] // LRU 最近最久未使用法
            keys.push(key) // 并将 key 放到后面[b,a]
        } else {
            cache[key] = vnode // 缓存 vnode
            keys.push(key) // 将 key 存入
            // prune oldest entry
            if (this.max && keys.length > parseInt(this.max)) { // 缓存的太多超过了 max 就需要删除掉
                pruneCacheEntry(cache, keys[0], keys, this._vnode) // 要删除第 0 个 但是现在渲染的就是第 0 个
            }
        }

        vnode.data.keepAlive = true // 并且标准 keep-alive 下的组件是一个缓存组件
```

```
        }
        return vnode || (slot && slot[0]) // 返回当前的虚拟节点
      }
    }
```

Vue.js 的 action 和 mutation 区别

mutation 是同步更新数据(内部会进行是否为异步方式更新数据的检测)

action 异步操作，可以获取数据后调佣 mutation 提交最终数据

Vue Router

Vue-Router 中导航守卫有哪些？

完整的导航解析流程

1. 导航被触发。
2. 在失活的组件里调用离开守卫。
3. 调用全局的 beforeEach 守卫。
4. 在重用的组件里调用 beforeRouteUpdate 守卫 (2.2+)。
5. 在路由配置里调用 beforeEnter。
6. 解析异步路由组件。
7. 在被激活的组件里调用 beforeRouteEnter。
8. 调用全局的 beforeResolve 守卫 (2.5+)。
9. 导航被确认。
10. 调用全局的 afterEach 钩子。
11. 触发 DOM 更新。

12. 用创建好的实例调用 beforeRouteEnter 守卫中传给 next 的回调函数。

说一下 Vue 的双向绑定数据的原理

vue.js 则是采用数据劫持结合发布者-订阅者模式的方式，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

Vue.js 实现数据双向绑定的原理 Object.defineProperty()

vue 实现数据双向绑定主要是：采用数据劫持结合发布者-订阅者模式的方式，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 将 MVVM 作为数据绑定的入口，整合 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（ vue 中是用来解析 {{}} ），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（ input ）—> 数据 model 变更双向绑定效果。

Vue.js 组件间的参数传递

父组件与子组件传值

父组件传给子组件：子组件通过 props 方法接受数据；

子组件传给父组件： $emit 方法传递参数

非父子组件间的数据传递，兄弟组件传值

eventBus ，就是创建一个事件中心，相当于中转站，可以用它来传递事件和接收事件。项目比较小时，用这个比较合适（虽然也有不少人推荐直接用VUEX ，具体来说看需求）

请列举出 3 个 Vue 中常用的生命周期钩子函数?

created：实例已经创建完成之后调用,在这一步,实例已经完成数据观测, 属性和方法的运 算, watch/event 事件回调. 然而, 挂载阶段还没有开始, $el 属性目前还不可

mounted：el 被新创建的 vm.$el 替换，并挂载到实例上去之后调用该钩子。如果 root 实例挂载了一个文档内元素，当 mounted 被调用时 vm.$el 也在文档内。

activated：keep-alive 组件激活时调用

在 beforeCreate 钩子函数调用的时候，是获取不到 props 或者 data 中的数据的， 因为这些数据的初始化都在 initState 中。

然后会执行 created 钩子函数，在这一步的时候已经可以访问到之前不能访问到的数 据，但是这时候组件还没被挂载，所以是看不到的。

接下来会先执行 beforeMount 钩子函数，开始创建 VDOM ，最后执行 mounted 钩 子，并将 VDOM 渲染为真实 DOM 并且渲染数据。组件中如果有子组件的话，会递归挂载 子组件，只有当所有子组件全部挂载完毕，才会执行根组件的挂载钩子。

接下来是数据更新时会调用的钩子函数 beforeUpdate 和 updated ，这两个钩子函数没 什么好说的，就是分别在数据更新前和更新后会调用。

另外还有 keep-alive 独有的生命周期，分别为 activated 和 deactivated 。用 keep-alive 包裹的组件在切换时不会进行销毁，而是缓存到内存中并执行 deactivated 钩子函数，命中缓存渲染后会执行 actived 钩子函数。

最后就是销毁组件的钩子函数 beforeDestroy 和 destroyed 。前者适合移除事件、定 时器等等，否则可能会引起内存泄露的问题。然后进行一系列的销毁操作，如果有子组件 的话，也会递归销毁子组件，所有子组件都销毁完毕后才会执行根组件的 destroyed 钩 子函数

vue-cli 工程技术集合介绍

问题一：构建的 vue-cli 工程都到了哪些技术，它们的作用分别是什么？

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。

vuex 等：一个专为 vue 设计的移动端 UI 组件库。

创建一个 emit.js 文件，用于 vue 事件机制的管理。

webpack ：模块加载和 vue-cli 工程打包器。

问题二：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
```

在浏览器上自动弹出一个 展示 vue-cli 工程打包后 app.js 、manifest.js 、 vendor.js 文件里面所包含代码的 面。可以具此优化 vue-cli 生产环境部署的静态资源，提升 面 的加载速度

实现 Vue SSR

其基本实现原理

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，还进行路由匹配与数据预获取。

webpack 为客服端打包一个 Client Bundle ，为服务端打包一个 Server Bundle 。

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 。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 stroe 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

Vue SSR 的实现 ,主要就是把 Vue 的组件输出成一个完整 HTML , vue-server-renderer 就是干这事的

Vue SSR 需要做的事多点（输出完整 HTML），除了 complier -> vnode ，还需如数据 获取填充至 HTML 、客户端混合（ hydration ）、缓存等等。 相比于其他模板引擎 （ ejs , jade 等），最终要实现的目的是一样的，性能上可能要差点

Vue.js 组件 data 为什么必须是函数

每个组件都是 Vue 的实例。

组件共享 data 属性，当 data 的值是同一个引用类型的值时，改变其中一个会影响其 他

Vue.js extend 能做什么

这个 API 很少用到，作用是扩展组件生成一个构造器，通常会与 $mount 一起使用。

```
// 创建组件构造器
let Component = Vue.extend({
    template: '<div>test</div>'
})
// 挂载到 #app 上
new Component().$mount('#app')
// 除了上面的方式，还可以用来扩展已有的组件
let SuperComponent = Vue.extend(Component)
new SuperComponent({
    created() {
```

```
        console.log(1)
    }
})
new SuperComponent().$mount('#app')
```

Vue.js mixin 和 mixins 区别

mixin 用于全局混入，会影响到每个组件实例，通常插件都是这样做初始化的

```
Vue.mixin({
    beforeCreate() {
        // ...逻辑
        // 这种方式会影响到每个组件的 beforeCreate 钩子函数
    }
})
```

虽然文档不建议我们在应用中直接使用 mixin ，但是如果不滥用的话也是很有帮助的， 比如可以全局混入封装好的 ajax 或者一些工具函数等等。

mixins 应该是我们最常使用的扩展组件的方式了。如果多个组件中有相同的业务逻辑， 就可以将这些逻辑剥离出来，通过 mixins 混入代码，比如上拉下拉加载数据这种逻辑等 等。

另外需要注意的是 mixins 混入的钩子函数会先于组件内的钩子函数执行 ,并且在遇到同 名选项的时候也会有选择性的进行合并，具体可以阅读 文档。

Vue.js computed 和 watch 区别

omputed 是计算属性，依赖其他属性计算值，并且 computed 的值有缓存，只有当计 算值变化才会返回内容。

watch 监听到值的变化就会执行回调，在回调中可以进行一些逻辑操作。

所以一般来说需要依赖别的属性来动态获得值的时候可以使用 computed ,对于监听到值 的变化需要做一些复杂业务逻辑的情况可以使用 watch 。

另外 computer 和 watch 还都支持对象的写法，这种方式知道的人并不多。

```
vm.$watch('obj', {
// 深度遍历
deep: true,
// 立即触发
immediate: true,
// 执行的函数
handler: function(val, oldVal) {}
})
var vm = new Vue({
    data: { a: 1 },
    computed: {
        aPlus: {
            // this.aPlus 时触发
            get: function () {
                return this.a + 1
            },
            // this.aPlus = 1 时触发
            set: function (v) {
                this.a = v - 1
            }
```

```
        }
    }
})
```

Vue.js keep-alive 组件有什么作用

如果你需要在组件切换的时候，保存一些组件的状态防止多次渲染，就可以使用 keep- alive 组件包裹需要保存的组件。

对于 keep-alive 组件来说，它拥有两个独有的生命周期钩子函数，分别为 activated 和 deactivated 。用 keep-alive 包裹的组件在切换时不会进行销毁，而是缓存到内 存中并执行 deactivated 钩子函数，命中缓存渲染后会执行 actived 钩子函数。

Vue.js 响应式原理

Vue 内部使用了 Object.defineProperty() 来实现数据响应式，通过这个函数可以监听到 set 和 get 的事件

```
var data = { name: 'poetries' }
observe(data)
let name = data.name // -> get value
data.name = 'yyy' // -> change value
function observe(obj) {
    // 判断类型
    if (!obj || typeof obj !== 'object') {
        return
    }
    Object.keys(obj).forEach(key => {
        defineReactive(obj, key, obj[key])
```

```
    })
}
function defineReactive(obj, key, val) {
    // 递归子属性
    observe(val)
    Object.defineProperty(obj, key, {
        // 可枚举
        enumerable: true,
        // 可配置
        configurable: true,
        // 自定义函数
        get: function reactiveGetter() {
            console.log('get value')
            return val
        },
        set: function reactiveSetter(newVal) {
            console.log('change value')
            val = newVal
        }
    })
}
```

以上代码简单的实现了如何监听数据的 set 和 get 的事件，但是仅仅如此 是不够的，因为自定义的函数一开始是不会执行的。只有先执行了依赖收集， 从能在属性更新的时候派发更新，所以接下来我们需要先触发依赖收集

```
<div>
    {{name}}
</div>
```

在解析如上模板代码时，遇到 就会进行依赖收集。

接下来我们先来实现一个 Dep 类，用于解耦属性的依赖收集和派发更新操作

```
// 通过 Dep 解耦属性的依赖和更新操作
class Dep {
constructor() {
    this.subs = []
}
// 添加依赖
addSub(sub) {
    this.subs.push(sub)
}
// 更新
notify() {
    this.subs.forEach(sub => {
```

```
            sub.update()
        })
    }
}
// 全局属性，通过该属性
```

以上的代码实现很简单，当需要依赖收集的时候调用 addSub ，当需要派发 更新的时候调用 notify 。

接下来我们先来简单的了解下 Vue 组件挂载时添加响应式的过程。在组件挂 载时，会先对所有需要的属性调用 Object.defineProperty() ，然后实例 化 Watcher ，传入组件更新的回调。在实例化过程中，会对模板中的属性进 行求值，触发依赖收集。

因为这一小节主要目的是学习响应式原理的细节，所以接下来的代码会简略的表达触发依赖收 集时的操作。

```
class Watcher {
    constructor(obj, key, cb) {
        // 将 Dep.target 指向自己
        // 然后触发属性的 getter 添加监听
        // 最后将 Dep.target 置空
        Dep.target = this
        this.cb = cb
        this.obj = obj
        this.key = key
        this.value = obj[key]
        Dep.target = null
    }
```

```
    update() {
        // 获得新值
        this.value = this.obj[this.key]
        // 调用 update 方法更新 Dom
        this.cb(this.value)
    }
}
```

以上就是 Watcher 的简单实现，在执行构造函数的时候将 Dep.target 指向自身，从而使得收集到了对应的 Watcher ，在派发更新的时候取出对应的 Watcher 然后执行 update 函数。

接下来，需要对 defineReactive 函数进行改造，在自定义函数中添加依赖收集和派发更新 相关的代码

```
function defineReactive(obj, key, val) {
    // 递归子属性
    observe(val)
    let dp = new Dep()
    Object.defineProperty(obj, key, {
        enumerable: true,
        configurable: true,
        get: function reactiveGetter() {
            console.log('get value')
            // 将 Watcher 添加到订阅
            if (Dep.target) {
                dp.addSub(Dep.target)
```

```
            }
            return val
        },
        set: function reactiveSetter(newVal) {
            console.log('change value')
            val = newVal
            // 执行 watcher 的 update 方法
            dp.notify()
        }
    })
}
```

以上所有代码实现了一个简易的数据响应式，核心思路就是手动触发一次属性 的 getter 来实现依赖收集。

现在我们就来测试下代码的效果，只需要把所有的代码复制到浏览器中执行，就会发现 面的 内容全部被替换了

```
var data = { name: 'poetries' }
observe(data)
function update(value) {
    document.querySelector('div').innerText = value
}
// 模拟解析到 `{{name}}` 触发的操作
new Watcher(data, 'name', update)
// update Dom innerText
```

```
data.name = 'yyy'
```

Object.defineProperty 的缺陷

以上已经分析完了 Vue 的响应式原理，接下来说一点 Object.defineProperty 中的缺 陷。

如果通过下标方式修改数组数据或者给对象新增属性并不会触发组件的重新渲染，因为 Object.defineProperty 不能拦截到这些操作，更精确的来说，对于数组而言，大部分 操作都是拦截不到的，只是 Vue 内部通过重写函数的方式解决了这个问题。 对于第一个问题， Vue 提供了一个 API 解决

```
export function set (target: Array<any> | Object, key: any, val: any): any
    // 判断是否为数组且下标是否有效
    if (Array.isArray(target) && isValidArrayIndex(key)) {
        // 调用 splice 函数触发派发更新
        // 该函数已被重写
        target.length = Math.max(target.length, key)
        target.splice(key, 1, val)
        return val
    }
    // 判断 key 是否已经存在
    if (key in target && !(key in Object.prototype)) {
        target[key] = val
        return val
    }
    const ob = (target: any).__ob__
    // 如果对象不是响应式对象，就赋值返回
```

```
    if (!ob) {
        target[key] = val
        return val
    }
    // 进行双向绑定
    defineReactive(ob.value, key, val)
    // 手动派发更新
    ob.dep.notify()
    return val
}
```

对于数组而言， Vue 内部重写了以下函数实现派发更新

```
// 获得数组原型
const arrayProto = Array.prototype
export const arrayMethods = Object.create(arrayProto)
// 重写以下函数
const methodsToPatch = [
    'push',
    'pop',
    'shift',
    'unshift',
    'splice',
    'sort',
```

```
    'reverse'
]
methodsToPatch.forEach(function (method) {
    // 缓存原生函数
    const original = arrayProto[method]
    // 重写函数
    def(arrayMethods, method, function mutator (...args) {
        // 先调用原生函数获得结果
        const result = original.apply(this, args)
        const ob = this.__ob__
        let inserted
        // 调用以下几个函数时，监听新数据
        switch (method) {
            case 'push':
            case 'unshift':
                inserted = args
                break
            case 'splice':
                inserted = args.slice(2)
                break
        }
        if (inserted) ob.observeArray(inserted)
```

```
        // 手动派发更新
        ob.dep.notify()
        return result
    })
})
```

编译过程

想必大家在使用 Vue 开发的过程中，基本都是使用模板的方式。那么你有过 「模板是怎么在浏览器中运行的」这种疑虑嘛？

首先直接把模板丢到浏览器中肯定是不能运行的，模板只是为了方便开发者进行开发。 Vue 会通过编译器将模板通过几个阶段最终编译为 render 函数，然后通过执行 render 函数生成 Virtual DOM 最终映射为真实 DOM 。

接下来我们就来学习这个编译的过程，了解这个过程中大概发生了什么事情。这个过程其 中又分为三个阶段，分别为：

- 将模板解析为 AST
- 优化 AST
- 将 AST 转换为 render 函数

在第一个阶段中，最主要的事情还是通过各种各样的正则表达式去匹配模板中的内容，然后将 内容提取出来做各种逻辑操作，接下来会生成一个最基本的 AST 对象

```
{
    // 类型
    type: 1,
    // 标签
    tag,
    // 属性列表
    attrsList: attrs,
```

```
        // 属性映射
        attrsMap: makeAttrsMap(attrs),
        // 父节点
        parent,
        // 子节点
        children: []
    }
```

然后会根据这个最基本的 AST 对象中的属性，进一步扩展 AST 。

当然在这一阶段中，还会进行其他的一些判断逻辑。比如说对比前后开闭标签是否一致， 判断根组件是否只存在一个，判断是否符合 HTML5 Content Model 规范等等问题。

接下来就是优化 AST 的阶段。在当前版本下， Vue 进行的优化内容其实还是不多的。只 是对节点进行了静态内容提取，也就是将永远不会变动的节点提取了出来，实现复用

Virtual DOM ，跳过对比算法的功能。在下一个大版本中， Vue 会在优化 AST 的阶段 继续发力，实现更多的优化功能，尽可能的在编译阶段压榨更多的性能，比如说提取静态 的属性等等优化行为。

最后一个阶段就是通过 AST 生成 render 函数了。其实这一阶段虽然分支有很多，但是 最主要的目的就是遍历整个 AST ，根据不同的条件生成不同的代码罢了。

NextTick 原理分析

nextTick 可以让我们在下次 DOM 更新循环结束之后执行延迟回调，用于 获得更新后的 DOM 。

在 Vue 2.4 之前都是使用的 microtasks ，但是 microtasks 的优先级过高，在某些 情况下可能会出现比事件冒泡更快的情况，但如果都使用 macrotasks 又可能会出现渲染 的性能问题。所以在新版本中，会默认使用 microtasks ，但在特殊情况下会使用 macrotasks ，比如 v-on 。

对于实现 macrotasks ，会先判断是否能使用 setImmediate ，不能的话降级为 MessageChannel ，以上都不行的话就使用 setTimeout

```
if (typeof setImmediate !== 'undefined' && isNative(setImmediate)) {
```

```
    macroTimerFunc = () => {
        setImmediate(flushCallbacks)
    }
} else if (
    typeof MessageChannel !== 'undefined' &&
    (isNative(MessageChannel) ||
    // PhantomJS
    MessageChannel.toString() === '[object MessageChannelConstructor]')
) {
    const channel = new MessageChannel()
    const port = channel.port2
    channel.port1.onmessage = flushCallbacks
    macroTimerFunc = () => {
        port.postMessage(1)
    }
} else {
    macroTimerFunc = () => {
        setTimeout(flushCallbacks, 0)
    }
}
```

以上代码很简单，就是判断能不能使用相应的 API

Vue.js 数据双向绑定原理,常见数据绑定的方案

Object.defineProperty（vue） ：劫持数据的 getter 和 setter

脏值检测（ angularjs ）：通过特定事件进行轮循 发布/订阅模式：通过消息发布并将消息进行订阅

Vue.js VDOM,三个 part

虚拟节点类，将真实 DOM 节点用 js 对象的形式进行展示，并提供 render 方法，将虚拟节点渲染成真实 DOM

节点 diff 比较：对虚拟节点进行 js 层面的计算，并将不同的操作都记录到 patch 对象

re-render ：解析 patch 对象，进行 re-render

补充 1：VDOM 的必要性？

创建真实 DOM 的代价高：真实的 DOM 节点 node 实现的属性很多，而 vnode 仅仅实 现一些必要的属性，相比起来，创建一个 vnode 的成本比较低。

触发多次浏览器重绘及回流：使用 vnode ，相当于加了一个缓冲，让一次数据变动所带 来的所有 node 变化，先在 vnode 中进行修改，然后 diff 之后对所有产生差异的节 点集中一次对 DOM tree 进行修改，以减少浏览器的重绘及回流。

补充 2：vue 为什么采用 vdom？

引入 Virtual DOM 在性能方面的考量仅仅是一方面。

性能受场景的影响是非常大的，不同的场景可能造成不同实现方案之间成倍的性能差距， 所以依赖细粒度绑定及 Virtual DOM 哪个的性能更好还真不是一个容易下定论的问题。

Vue 之所以引入了 Virtual DOM ，更重要的原因是为了解耦 HTML 依赖，这带来两个 非常重要的好处是：

不再依赖 HTML 解析器进行模版解析，可以进行更多的 AOT 工作提高运 行时效率：通过模版 AOT 编译， Vue 的运行时体积可以进一步压缩，运 行时效率可以进一步提升；

可以渲染到 DOM 以外的平台，实现 SSR 、同构渲染这些高级特性， Weex 等框架应用的就是这一特性。

综上， Virtual DOM 在性能上的收益并不是最主要的，更重要的是它使得 Vue 具备了现代框架应有的高级特性。

Vue 原理

谈一下你对 Vue.js 的 MVVM 原理的理解

传 统 MVC

传统的 MVC 指的是

M 是指业务模型，Model（模型）

V 是指用户界面，View（视图）

C 则是控制器，Controller（控制器）

使用 MVC 的目的是将 M 和 V 的实现代码分离，从而使同一个程序可以使用不同的表现形式。

MVVM

MVVM 是 Model-View-ViewModel 的缩写

Model 代表数据模型，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成 UI 展现出来。

ViewModel 监听模型数据的改变和控制视图行为、处理用户交互，简单理解就是一个同步 View 和 Model 的对象，连接 Model 和 View

传统的前端会将数据手动渲染到页面上,MVVM 模式不需要用户手动操作 dom 元素；

将数据绑定到 viewModel 层上，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。

ViewModel 就是我们 MVVM 模式中的桥梁.

六星教育 WEB 前端面试圣经

在 MVVM 架构下， View 和 Model 之间并没有直接的联系，而是通过 ViewModel 进行交互，Model 和 ViewModel 之间的交互是双向的，因 此 View 数据的变化会同步到 Model 中 ,而Model 数据的变化也会立即反 应到 View 上。

ViewModel 通过双向数据绑定把 View 层和 Model 层连接了起来，而 View 和 Model 之间的同步工作完全是自动的，无需人为干涉，因此开 发者只需关注业务逻辑，不需要手动操作 DOM，不需要关注数据状态的同 步问题，复杂的数据状态维护完全由 MVVM 来统一管理

请说一下 Vue.js 响应式数据的原理

1. 核心点:`Object.defineProperty`

2. 默认 `Vue` 在初始化数据时，会给 `data` 中的属性使用 `Object.defineProperty` 重新定义所有属性,当页面取到对应属性时。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。

对 应 源 码

```
Object.defineProperty(obj, key, {
    enumerable: true,
    configurable: true,
    get: function reactiveGetter () {
      const value = getter ? getter.call(obj) : val
      if (Dep.target) {
        dep.depend() // ** 收集依赖 ** /
        if (childOb) {
          childOb.dep.depend()
          if (Array.isArray(value)) {
```

```
                dependArray(value)
            }
        }
    }
    return value
  },
  set: function reactiveSetter (newVal) {
    const value = getter ? getter.call(obj) : val
    if (newVal === value || (newVal !== newVal && value !== value)) {
      return
    }
    if (process.env.NODE_ENV !== 'production' && customSetter) {
      customSetter()
    }
    val = newVal
    childOb = !shallow && observe(newVal)
    dep.notify() /**通知相关依赖进行更新**/
  }
})
```

Vue.js 的 nextTick 的实现原理?

理解:(**宏任务**和**微任务**) 异步方法

`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` 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

对应源码

```
let timerFunc    // 会定义一个异步方法
if (typeof Promise !== 'undefined' && isNative(Promise)) {    // promise
  const p = Promise.resolve()
  timerFunc = () => {
    p.then(flushCallbacks)
    if (isIOS) setTimeout(noop)
  }
  isUsingMicroTask = true
} else if (!isIE && typeof MutationObserver !== 'undefined' && ( // MutationObserver
  isNative(MutationObserver) ||
  MutationObserver.toString() === '[object MutationObserverConstructor]'
)) {
  let counter = 1
  const observer = new MutationObserver(flushCallbacks)
  const textNode = document.createTextNode(String(counter))
  observer.observe(textNode, {
    characterData: true
  })
```

```
    timerFunc = () => {
      counter = (counter + 1) % 2
      textNode.data = String(counter)
    }
    isUsingMicroTask = true
  } else if (typeof setImmediate !== 'undefined' ) { // setImmediate
    timerFunc = () => {
      setImmediate(flushCallbacks)
    }
  } else {
    timerFunc = () => {     // setTimeout
      setTimeout(flushCallbacks, 0)
    }
  }
  // nextTick 实现
  export function nextTick (cb?: Function, ctx?: Object) {
    let _resolve
    callbacks.push(() => {
      if (cb) {
        try {
          cb.call(ctx)
        } catch (e) {
```

```
            handleError(e, ctx, 'nextTick')
        }
    } else if (_resolve) {
        _resolve(ctx)
    }
  })
  if (!pending) {
    pending = true
    timerFunc()
  }
}
```

Vue.js 中模板编译原理

将 template 转化成 render 函数

对 应 源 码

```
function baseCompile (
  template: string,
  options: CompilerOptions
) {
  const ast = parse(template.trim(), options) // 1.将模板转化成 ast 语法树
  if (options.optimize !== false) {               // 2.优化树
```

```
        optimize(ast, options)
      }
      const code = generate(ast, options)          // 3.生成树
      return {
        ast,
        render: code.render,
        staticRenderFns: code.staticRenderFns
      }
    })
    const ncname = `[a-zA-Z_][\\-\\.0-9_a-zA-Z]*`;
    const qnameCapture = `((?:${ncname}\\:)?${ncname})`;
    const startTagOpen = new RegExp(`^<${qnameCapture}`); // 标签开头的正则 捕获的内容是标签名
    const endTag = new RegExp(`^<\\/${qnameCapture}[^>]*>`); // 匹配标签结尾的 </div>
    const attribute = /^\s*([^\s"'<>\/=]+)(?:\s*(=)\s*(?:"([^"]*)"+|'([^']*)'+|([^\s"'=<>`]+)))?/; // 匹配属性的
    const startTagClose = /^\s*(\/?)>/; // 匹配标签结束的 >
    let root;
    let currentParent;
    let stack = []
    function createASTElement(tagName,attrs){
      return {
```

```
            tag:tagName,
            type:1,
            children:[],
            attrs,
            parent:null
        }
    }
    function start(tagName,attrs){
        let element = createASTElement(tagName,attrs);
        if(!root){
            root = element;
        }
        currentParent = element;
        stack.push(element);
    }
    function chars(text){
        currentParent.children.push({
            type:3,
            text
        })
    }
    function end(tagName){
```

```
    const element = stack[stack.length-1];
    stack.length --;
    currentParent = stack[stack.length-1];
    if(currentParent){
        element.parent = currentParent;
        currentParent.children.push(element)
    }
}
function parseHTML(html){
    while(html){
        let textEnd = html.indexOf('<');
        if(textEnd == 0){
            const startTagMatch = parseStartTag();
            if(startTagMatch){
                start(startTagMatch.tagName,startTagMatch.attrs);
                continue;
            }
            const endTagMatch = html.match(endTag);
            if(endTagMatch){
                advance(endTagMatch[0].length);
                end(endTagMatch[1])
            }
```

```
        }
        let text;
        if(textEnd >=0 ){
            text = html.substring(0,textEnd)
        }
        if(text){
            advance(text.length);
            chars(text);
        }
    }
    function advance(n) {
        html = html.substring(n);
    }
    function parseStartTag(){
        const start = html.match(startTagOpen);
        if(start){
            const match = {
                tagName:start[1],
                attrs:[]
            }
            advance(start[0].length);
            let attr,end
```

```
            while(!(end = html.match(startTagClose)) && (attr=html.match(attribute))){
                advance(attr[0].length);
                match.attrs.push({name:attr[1],value:attr[3]})
            }
            if(end){
                advance(end[0].length);
                return match
            }
        }
    }
}
// 生成语法树
parseHTML(`<div id="container"><p>hello<span>zf</span></p></div>`);
function gen(node){
    if(node.type == 1){
        return generate(node);
    }else{
        return `_v(${JSON.stringify(node.text)})`
    }
}
function genChildren(el){
    const children = el.children;
```

```
    if(el.children){
        return `[${children.map(c=>gen(c)).join(',')}]`
    }else{
        return false;
    }
}
function genProps(attrs){
    let str = '';
    for(let i = 0; i < attrs.length;i++){
        let attr = attrs[i];
        str+= `${attr.name}:${attr.value},`;
    }
    return `{attrs:{${str.slice(0,-1)}}}`
}
function generate(el){
    let children = genChildren(el);
    let code = `_c('${el.tag}'${
        el.attrs.length? `,${genProps(el.attrs)}`:''
    }${
        children? `,${children}`:''
    })`;
    return code;
```

```
}
// 根据语法树生成新的代码
let code = generate(root);
let render = `with(this){return ${code}}`;

// 包装成函数
let renderFn = new Function(render);
console.log(renderFn.toString());
```

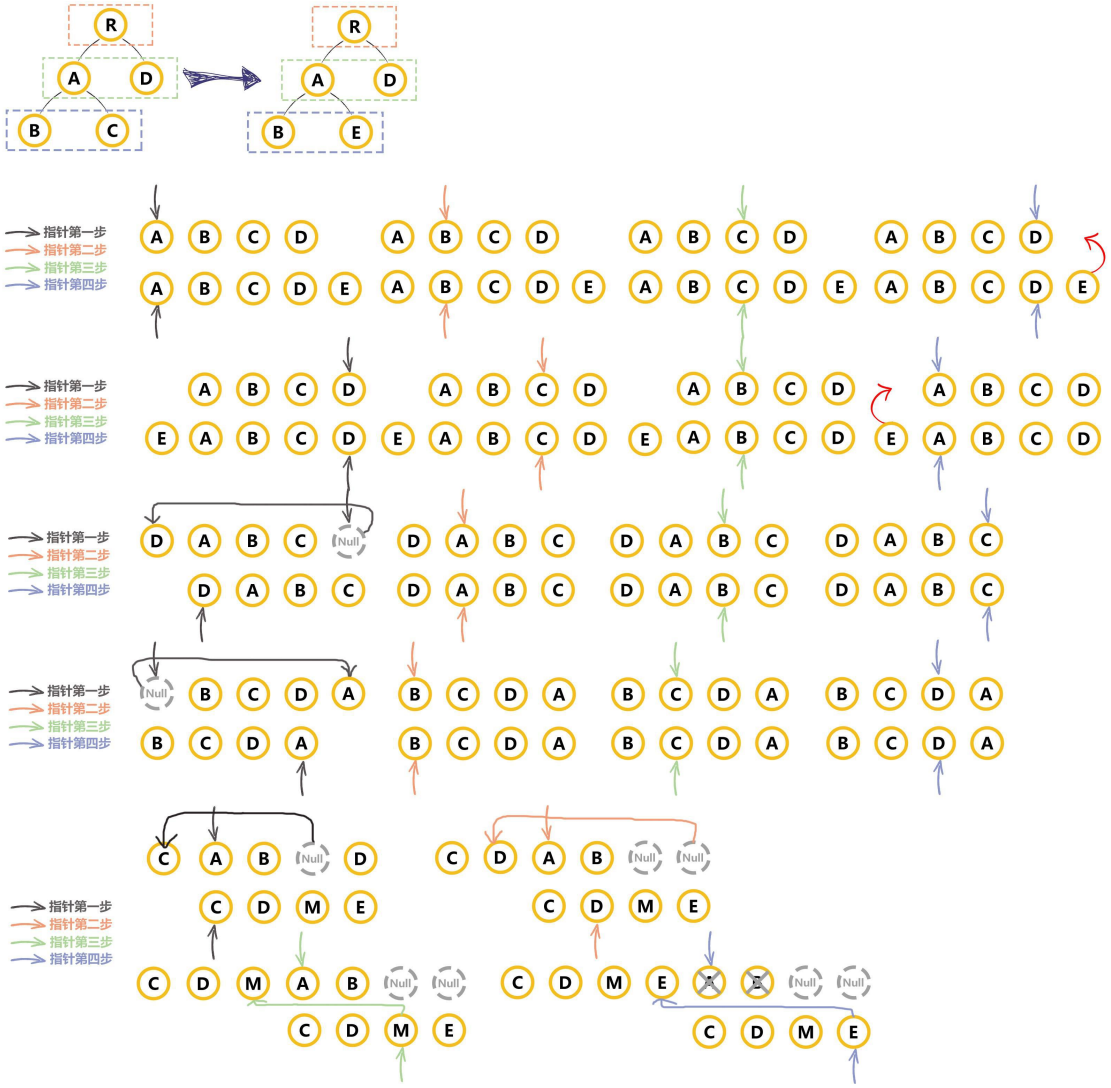
简述 Vue.js 中 diff 算法原理

1.先同级比较，在比较子节点

2.先判断一方有儿子一方没儿子的情况

3.比较都有儿子的情况

4.递归比较子节点



对应源码

core/vdom/patch.js

```
const oldCh = oldVnode.children // 老的儿子
const ch = vnode.children   // 新的儿子
if (isUndef(vnode.text)) {
    if (isDef(oldCh) && isDef(ch)) {
        // 比较孩子
```

```
            if (oldCh !== ch) updateChildren(elm, oldCh, ch, insertedVnodeQueue,
removeOnly)
        } else if (isDef(ch)) { // 新的儿子有 老的没有
            if (isDef(oldVnode.text)) nodeOps.setTextContent(elm, '')
            addVnodes(elm, null, ch, 0, ch.length - 1, insertedVnodeQueue)
        } else if (isDef(oldCh)) { // 如果老的有新的没有 就删除
            removeVnodes(oldCh, 0, oldCh.length - 1)
        } else if (isDef(oldVnode.text)) {   // 老的有文本 新的没文本
            nodeOps.setTextContent(elm, '') // 将老的清空
        }
    } else if (oldVnode.text !== vnode.text) { // 文本不相同替换
        nodeOps.setTextContent(elm, vnode.text)
    }
    function updateChildren (parentElm, oldCh, newCh, insertedVnodeQueue, removeOnly) {
        let oldStartIdx = 0
        let newStartIdx = 0
        let oldEndIdx = oldCh.length - 1
        let oldStartVnode = oldCh[0]
        let oldEndVnode = oldCh[oldEndIdx]
        let newEndIdx = newCh.length - 1
        let newStartVnode = newCh[0]
        let newEndVnode = newCh[newEndIdx]
```

```
    let oldKeyToIdx, idxInOld, vnodeToMove, refElm

    // removeOnly is a special flag used only by <transition-group>
    // to ensure removed elements stay in correct relative positions
    // during leaving transitions
    const canMove = !removeOnly

    if (process.env.NODE_ENV !== 'production') {
      checkDuplicateKeys(newCh)
    }

    while (oldStartIdx <= oldEndIdx && newStartIdx <= newEndIdx) {
      if (isUndef(oldStartVnode)) {
        oldStartVnode = oldCh[++oldStartIdx] // Vnode has been moved left
      } else if (isUndef(oldEndVnode)) {
        oldEndVnode = oldCh[--oldEndIdx]
      } else if (sameVnode(oldStartVnode, newStartVnode)) {
        patchVnode(oldStartVnode, newStartVnode, insertedVnodeQueue, newCh, newStartIdx)
        oldStartVnode = oldCh[++oldStartIdx]
        newStartVnode = newCh[++newStartIdx]
      } else if (sameVnode(oldEndVnode, newEndVnode)) {
```

```
patchVnode(oldEndVnode, newEndVnode, insertedVnodeQueue, newCh, newEndIdx)
oldEndVnode = oldCh[--oldEndIdx]
newEndVnode = newCh[--newEndIdx]
} else if (sameVnode(oldStartVnode, newEndVnode)) { // Vnode moved right
patchVnode(oldStartVnode, newEndVnode, insertedVnodeQueue, newCh, newEndIdx)
canMove && nodeOps.insertBefore(parentElm, oldStartVnode.elm, nodeOps.nextSibling(oldEndVnode.elm))
oldStartVnode = oldCh[++oldStartIdx]
newEndVnode = newCh[--newEndIdx]
} else if (sameVnode(oldEndVnode, newStartVnode)) { // Vnode moved left
patchVnode(oldEndVnode, newStartVnode, insertedVnodeQueue, newCh, newStartIdx)
canMove && nodeOps.insertBefore(parentElm, oldEndVnode.elm, oldStartVnode.elm)
oldEndVnode = oldCh[--oldEndIdx]
newStartVnode = newCh[++newStartIdx]
} else {
if (isUndef(oldKeyToIdx)) oldKeyToIdx = createKeyToOldIdx(oldCh, oldStartIdx, oldEndIdx)
idxInOld = isDef(newStartVnode.key)
? oldKeyToIdx[newStartVnode.key]
: findIdxInOld(newStartVnode, oldCh, oldStartIdx, oldEndIdx)
if (isUndef(idxInOld)) { // New element
```

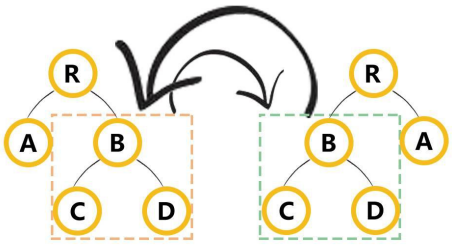
```
createElm(newStartVnode, insertedVnodeQueue, parentElm, oldStartVnode.elm, false, newCh, newStartIdx)
} else {
vnodeToMove = oldCh[idxInOld]
if (sameVnode(vnodeToMove, newStartVnode)) {
patchVnode(vnodeToMove, newStartVnode, insertedVnodeQueue, newCh, newStartIdx)
oldCh[idxInOld] = undefined
canMove && nodeOps.insertBefore(parentElm, vnodeToMove.elm, oldStartVnode.elm)
} else {
// same key but different element. treat as new element
createElm(newStartVnode, insertedVnodeQueue, parentElm, oldStartVnode.elm, false, newCh, newStartIdx)
}
}
newStartVnode = newCh[++newStartIdx]
}
}
if (oldStartIdx > oldEndIdx) {
refElm = isUndef(newCh[newEndIdx + 1]) ? null : newCh[newEndIdx + 1].elm
addVnodes(parentElm, refElm, newCh, newStartIdx, newEndIdx, insertedVnodeQueue)
} else if (newStartIdx > newEndIdx) {
```

```
        removeVnodes(oldCh, oldStartIdx, oldEndIdx)
    }
  }
```

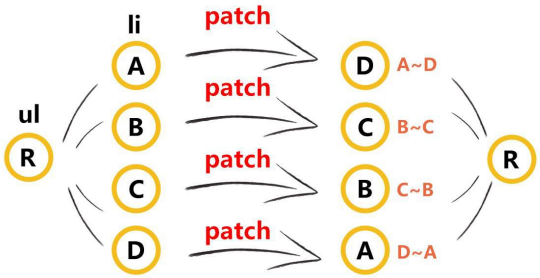
Vue.js 中 v-for 中为什么要用 key

看图

有Key的情况下：



无Key的情况下：



Vue.js 中事件绑定的原理

1.原生 dom 事件的绑定,采用的是 addEventListener 实现

2.组件绑定事件采用的是$on 方法

对 应 源 码

- 事件的编译：
-
-

```
let compiler = require('vue-template-compiler');
let r1 = compiler.compile('<div @click="fn()"></div>');
let r2 = compiler.compile('<my-component @click.native="fn" @click="fn1"></my-component>');
console.log(r1); // {on:{click}}
```

```
5.                  console.log(r2); // {nativeOnOn:{click},on:{click}}
.
  passive: boolean
.         ) {
.           target.addEventListener( // 给当前的 dom 添加事件
.             name,
.             handler,
.             supportsPassive
.               ? { capture, passive }
.               : capture
.           )
.         }
```

vue 中绑定事件是直接绑定给真实 dom 元素的

2.组件中绑定事件

```
1.                  export function updateComponentListeners (
2.                  vm: Component,
3.                  listeners: Object,
4.                  oldListeners: ?Object
5.                  ) {
```

```
6.          target = vm
7.          updateListeners(listeners, oldListeners || {}, add, remove, createOnceHandler,
vm)
8.          target = undefined
9.          }
10.         function add (event, fn) {
11.         target.$on(event, fn)
12.         }
13.
```

组件绑定事件是通过 vue 中自定义的$on 方法来实现的

Vue 组件

Vue.js 组件的生命周期

总共分为 8 个阶段创建前/后，载入前/后，更新前/后，销毁前/后

创建前/后： 在 beforeCreate 阶段， vue 实例的挂载元素 el 和数据对象 data 都为 undefined ，还未初始化。在 created 阶段， vue 实例的数据对象 data 有了，el 还 没有

载入前/后：在 beforeMount 阶段， vue 实例的 $el 和 data 都初始化了，但还是挂载之前为虚拟的 dom 节点， data.message 还未替换。在 mounted 阶段， vue 实例挂载完成，data.message 成功渲染。

更新前/后：当 data 变化时，会触发 beforeUpdate 和 updated 方法

销毁前/后：在执行 destroy 方法后，对 data 的改变不会再触发周期函数，说明此时 vue 实例已经解除了事件监听以及和 dom 的绑定，但是 dom 结构依然存在

要掌握每个生命周期什么时候被调用

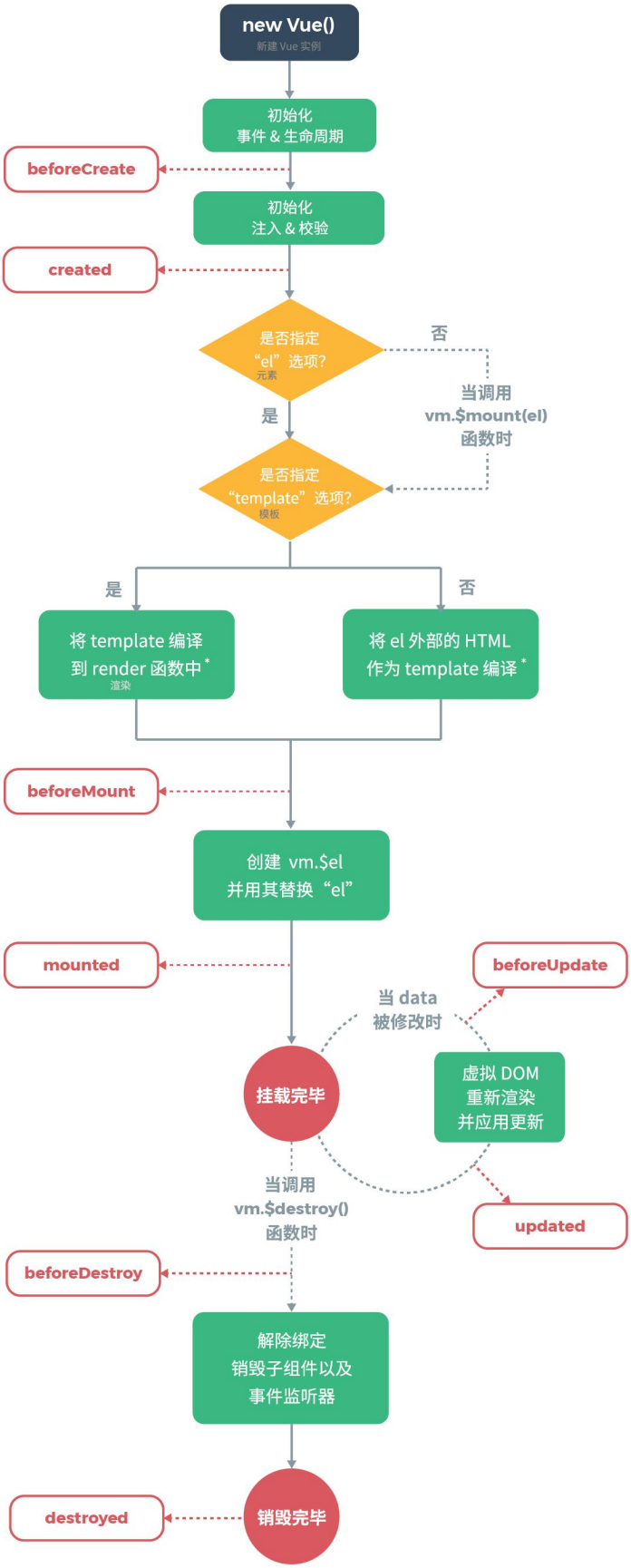
- beforeCreate 在实例初始化之后，数据观测(data observer) 之前被调用。

- created 实例已经创建完成之后被调用。在这一步，实例已完成以下的配置：数据观测(data observer)，属性和方法的运算， watch/event 事件回调。这里没有$el
- beforeMount 在挂载开始之前被调用：相关的 render 函数首次被调用。
- mounted el 被新创建的 vm.$el 替换，并挂载到实例上去之后调用该钩子。
- beforeUpdate 数据更新时调用，发生在虚拟 DOM 重新渲染和打补丁之前。
- updated 由于数据更改导致的虚拟 DOM 重新渲染和打补丁，在这之后会调用该钩子。
- beforeDestroy 实例销毁之前调用。在这一步，实例仍然完全可用。
- destroyed Vue 实例销毁后调用。调用后，Vue 实例指示的所有东西都会解绑定，所有的事件监听器会被移除，所有的子实例也会被销毁。 该钩子在服务器端渲染期间不被调用。

要掌握每个生命周期内部可以做什么事

- created 实例已经创建完成，因为它是最早触发的原因可以进行一些数据，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地更改状态，这不会触发附加的重渲染过程。
- updated 可以执行依赖于 DOM 的操作。然而在大多数情况下，你应该避免在此期间更改状态，因为这可能会导致更新无限循环。 该钩子在服务器端渲染期间不被调用。
- destroyed 可以执行一些优化操作,清空定时器，解除绑定事件

六星教育 WEB 前端面试圣经



* 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

六星教育 WEB 前端面试圣经

引申的问题

什么是 vue生命周期？

答： Vue 实例从创建到销毁的过程，就是生命周期。从开始创建、初始化数据、编译模板、挂载 Dom →渲染、更新→渲染、销毁等一系列过程，称之为 Vue 的生命周期。

vue生命周期的作用是什么？

答：它的生命周期中有多个事件钩子，让我们在控制整个 Vue 实例的过程时更容易形成好的 逻辑。

vue生命周期总共有几个阶段？

答：它可以总共分为 8 个阶段：创建前/后、载入前/后、更新前/后、销毁前/销毁后。

第一次 面加载会触发哪几个钩子？

答：会触发下面这几个 beforeCreate 、 created 、 beforeMount 、 mounted 。

DOM 渲染在哪个周期中就已经完成？

答： DOM 渲染在 mounted 中就已经完成了

描述 Vue.js 组件渲染和更新过程

渲染组件时，会通过 `Vue.extend` 方法构建子组件的构造函数，并进行实例化。最终手动调用 `$mount()` 进行挂载。更新组件时会进行 `patchVnode` 流程.核心就是 diff 算法

Vue.js 组件中的 data 为什么是一个函数?

同一个组件被复用多次，会创建多个实例。

这些实例用的是同一个构造函数，如果 `data` 是一个对象的话。那么所有组件都共享了同一个对象。为了保证组件的数据独立性要求每个组件必须通过 `data` 函数返回一个对象作为组件的状态。

对应源码

`core/global-api/extend.js line:33`

```
Sub.options = mergeOptions(
```

```
        Super.options,
        extendOptions
    )
    function mergeOptions(){
      function mergeField (key) {
            const strat = strats[key] || defaultStrat
            options[key] = strat(parent[key], child[key], vm, key)
        }
    }
    strats.data = function (
      parentVal: any,
      childVal: any,
      vm?: Component
    ): ?Function {
      if (!vm) { // 合并是会判断子类的 data 必须是一个函数
        if (childVal && typeof childVal !== 'function') {
            process.env.NODE_ENV !== 'production' && warn(
              'The "data" option should be a function ' +
              'that returns a per-instance value in component ' +
              'definitions.',
              vm
            )
```

```
        return parentVal
      }
      return mergeDataOrFn(parentVal, childVal)
    }
    return mergeDataOrFn(parentVal, childVal, vm)
  }
```

一个组件被使用多次，用的都是同一个构造函数。为了保证组件的不同的实例 data 不冲突，要求 data 必须是一个函数，这样组件间不会相互影响

Vue.js 父子组件生命周期调用顺序

过 程 理 解

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

组件的调用顺序都是先父后子,渲染完成的顺序肯定是先子后父

组件的销毁操作是先父后子，销毁完成的顺序是先子后父

Vue.js 组件如何通信以及有哪些方式?

父子间通信 父->子通过 props、子-> 父$on、$emit

获取父子组件实例的方式$parent、$children

在父组件中提供数据子组件进行消费 Provide、inject

Ref 获取实例的方式调用组件的属性或者方法

Event Bus 实现跨组件通信

Vuex 状态管理实现通信

1. 父子通信

父组件通过 props 传递数据给子组件，子组件通过 emit 发送事件传递数据给父组件， 这两种方式是最常用的父子通信实现办法。

这种父子通信方式也就是典型的单向数据流，父组件通过 props 传递数据，子组件不能 直接修改 props ，而是必须通过发送事件的方式告知父组件修改数据。

另外这两种方式还可以使用语法糖 v-model 来直接实现，因为 v-model 默认会解析成 名为 value 的 prop 和名为 input 的事件。这种语法糖的方式是典型的双向绑定， 常用于 UI 控件上，但是究其根本，还是通过事件的方法让父组件修改数据。

当然我们还可以通过访问 $parent 或者 $children 对象来访问组件实例中的方法和数 据。

另外如果你使用 Vue 2.3 及以上版本的话还可以使用 $listeners 和 .sync 这两个属 性。

$listeners 属性会将父组件中的 (不含 .native 修饰器的) v-on 事件监听器传递给 子组件，子组件可以通过访问 $listeners 来自定义监听器。

.sync 属性是个语法糖，可以很简单的实现子组件与父组件通信

```
<!--父组件中-->
<input :value.sync="value" />
```

```
<!--以上写法等同于-->
<input :value="value" @update:value="v => value = v"></comp>
<!--子组件中-->
<script>
this.$emit('update:value', 1)
</script>
```

2. 兄弟组件通信

对于这种情况可以通过查找父组件中的子组件实现，也就是 this.$parent.$children ，在 $children 中可以通过组件 name 查询 到需要的组件实例，然后进行通信。

3. 跨多层次组件通信

对于这种情况可以使用 Vue 2.2 新增的 API provide / inject ，虽然文 档中不推荐直接使用在业务中，但是如果用得好的话还是很有用的。

假设有父组件 A ，然后有一个跨多层级的子组件 B

```
// 父组件 A
export default {
    provide: {
        data: 1
    }
}
// 子组件 B
export default {
    inject: ['data'],
```

```
    mounted() {
        // 无论跨几层都能获得父组件的 data 属性
        console.log(this.data) // => 1
    }
}
```

终极办法解决一切通信问题

只要你不怕麻烦，可以使用 Vuex 或者 Event Bus 解决上述所有的通信情 况。

Vue.js 怎么快速定位哪个组件出现性能问题

如果组件功能多打包出的结果会变大 ,我可以采用异步的方式来加载组件。主要依赖 import()这个语法，可以实现文件的分割加载。

```
components:{
AddCustomerSchedule(resolve) {
    require(["../components/AddCustomer"], resolve);
}
}
```

对 应 源 码

```
export function createComponent (
  Ctor: Class<Component> | Function | Object | void,
```

```
    data: ?VNodeData,
    context: Component,
    children: ?Array<VNode>,
    tag?: string
  ): VNode | Array<VNode> | void {

    // async component
    let asyncFactory
    if (isUndef(Ctor.cid)) {
      asyncFactory = Ctor
      Ctor = resolveAsyncComponent(asyncFactory, baseCtor) // 默认调用此函数时返回 undefiend
      // 第二次渲染时 Ctor 不为 undefined
      if (Ctor === undefined) {
        return createAsyncPlaceholder( // 渲染占位符 空虚拟节点
          asyncFactory,
          data,
          context,
          children,
          tag
        )
      }
```

```
    }
  }
  function resolveAsyncComponent (
    factory: Function,
    baseCtor: Class<Component>
  ): Class<Component> | void {
    if (isDef(factory.resolved)) { // 3.在次渲染时可以拿到获取的最新组件
      return factory.resolved
    }
    const resolve = once((res: Object | Class<Component>) => {
      factory.resolved = ensureCtor(res, baseCtor)
      if (!sync) {
        forceRender(true) //2. 强制更新视图重新渲染
      } else {
        owners.length = 0
      }
    })
    const reject = once(reason => {
      if (isDef(factory.errorComp)) {
        factory.error = true
        forceRender(true)
      }
```

```
    })
    const res = factory(resolve, reject)// 1.将 resolve 方法和 reject 方法传入 ,用户调用 resolve 方法后
    sync = false
    return factory.resolved
}
```

优化和改进

Vue.js ajax 请求放在哪个生命周期中

在 created 的时候，视图中的 dom 并没有渲染出来，所以此时如果直接去操 dom 节点，无法找到相关的元素

在 mounted 中，由于此时 dom 已经渲染出来了，所以可以直接操作 dom 节点

一般情况下都放到 mounted 中,保证逻辑的统一性,因为生命周期是同步执行的，ajax 是异步执行的

服务端渲染不支持 mounted 方法，所以在服务端渲染的情况下统一放到 created 中

Vue.js 中相同逻辑如何抽离？

Vue.mixin 用法 给组件每个生命周期，函数等都混入一些公共逻辑

对 应 源 码

```
Vue.mixin = function (mixin: Object) {
    this.options = mergeOptions(this.options, mixin); // 将当前定义的属性合并到每个组件中
    return this
}
export function mergeOptions (
```

```
  parent: Object,
  child: Object,
  vm?: Component
): Object {
  if (!child._base) {
    if (child.extends) { // 递归合并 extends
      parent = mergeOptions(parent, child.extends, vm)
    }
    if (child.mixins) { // 递归合并 mixin
      for (let i = 0, l = child.mixins.length; i < l; i++) {
        parent = mergeOptions(parent, child.mixins[i], vm)
      }
    }
  }
  const options = {} // 属性及生命周期的合并
  let key
  for (key in parent) {
    mergeField(key)
  }
  for (key in child) {
    if (!hasOwn(parent, key)) {
      mergeField(key)
```

```
    }
  }
  function mergeField (key) {
    const strat = strats[key] || defaultStrat
    // 调用不同属性合并策略进行合并
    options[key] = strat(parent[key], child[key], vm, key)
  }
  return options
}
```

Vue.js 中常见性能优化

1.编码优化：

1.不要将所有的数据都放在 data 中，data 中的数据都会增加 getter 和 setter，会收集对应的 watcher

2.vue 在 v-for 时给每项元素绑定事件需要用事件代理

3.SPA 页面采用 keep-alive 缓存组件

4.拆分组件(提高复用性、增加代码的可维护性,减少不必要的渲染)

5.v-if 当值为 false 时内部指令不会执行,具有阻断功能，很多情况下使用 v-if 替代 v-show

6.key 保证唯一性 （ 默认 vue 会采用就地复用策略 ）

7.Object.freeze 冻结数据

8.合理使用路由懒加载、异步组件

9.尽量采用 runtime 运行时版本

10.数据持久化的问题 （ 防抖、节流 ）

2. Vue 加载性能优化：

- 第三方模块按需导入（babel-plugin-component）
- 滚动到可视区域动态加载（ https://tangbc.github.io/vue-virtual-scroll-list ）
- 图片懒加载（https://github.com/hilongjw/vue-lazyload.git）

3.用户体验：

- app-skeleton 骨架屏
- app-shellapp 壳
- pwa

4.SEO 优化：

- 预渲染插件 prerender-spa-plugin
- 服务端渲染 ssr

5.打包优化：

- 使用 cdn 的方式加载第三方模块
- 多线程打包 happypack
- splitChunks 抽离公共文件
- sourceMap 生成

6.缓存，压缩

- 客户端缓存、服务端缓存
- 服务端 gzip 压缩

Vue3.0 你知道有哪些改进?

六星教育 WEB 前端面试圣经

Vue3 采用了 TS 来编写

支持 Composition API

Vue3 中响应式数据原理改成 proxy

vdom 的对比算法更新，只更新 vdom 的绑定了动态数据的部分