




微信号：zhufengjiagou

加微信可免费领取更多精彩视频

珠峰前端架构直播课(每周)

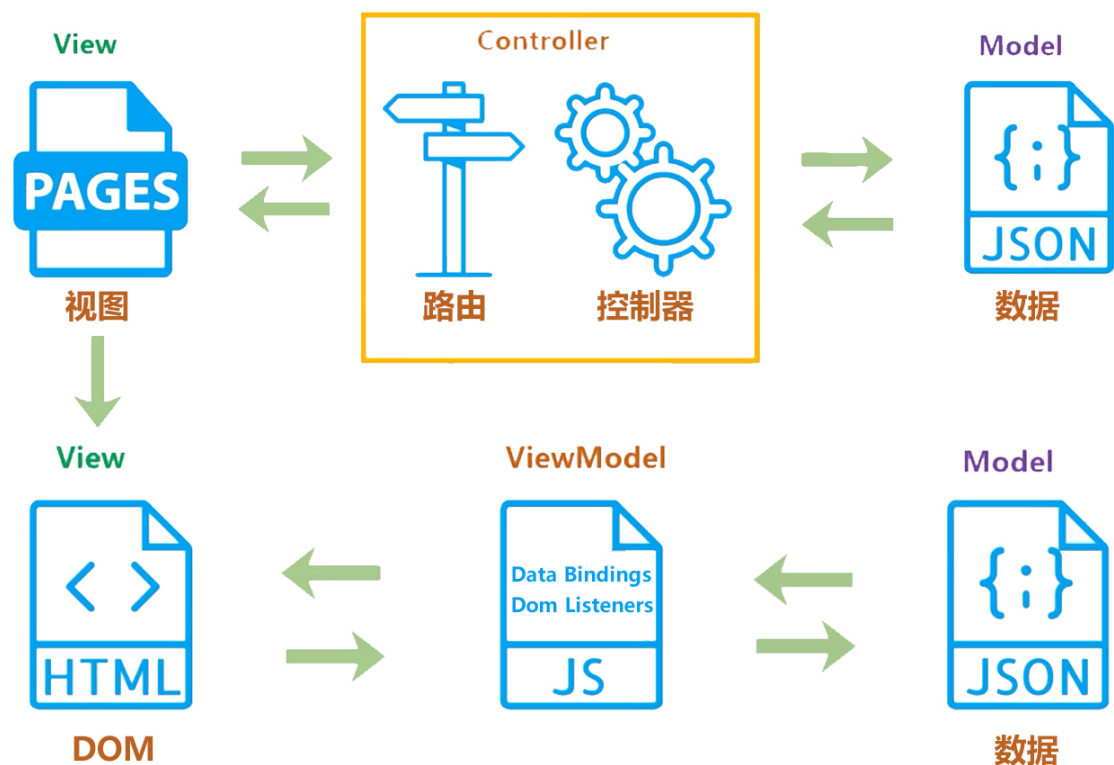
- 晋级高级前端-对标阿里P6+
- 前端技术专家联合研发

JS高级 / VUE / REACT / NodeJS / 前端工程化 / 项目实战 / 测试 / 运维



长按识别二维码加微信
获取直播地址和历史精彩视频

1.谈一下你对MVVM原理的理解



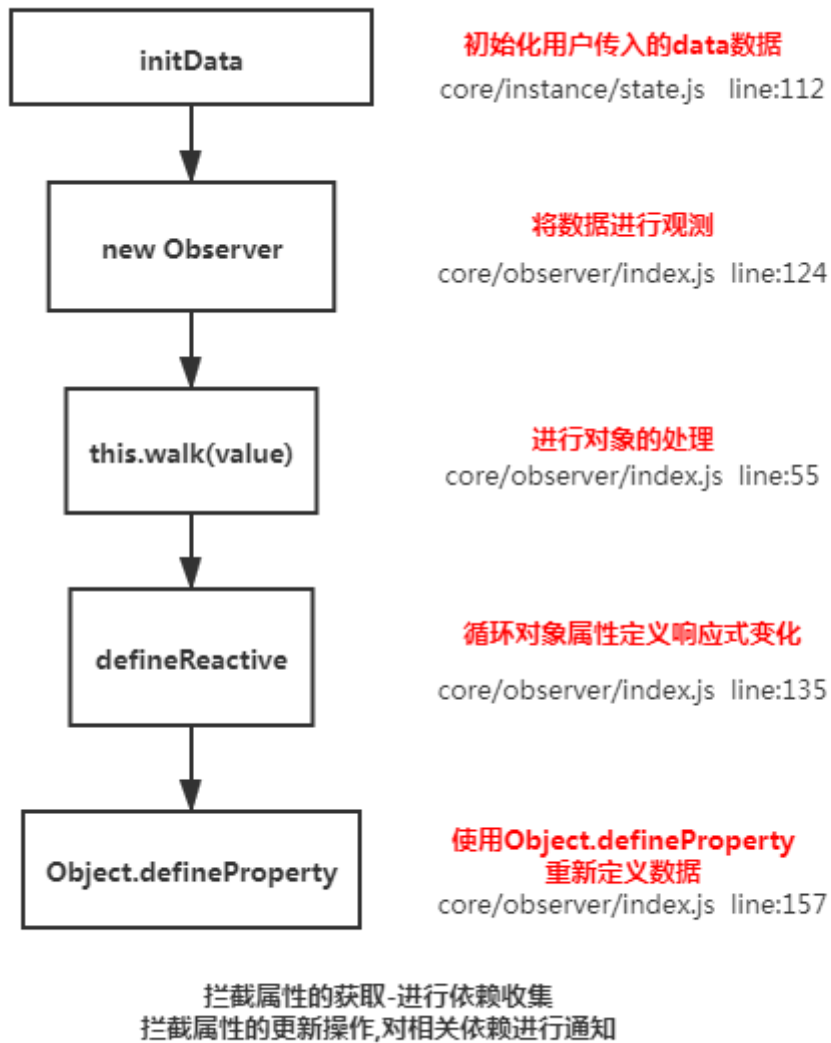
- 传统的 MVC 指的是,用户操作会请求服务端路由,路由会调用对应的控制器来处理,控制器会获取数据。将结果返回给前端,页面重新渲染
- MVVM:传统的前端会将数据手动渲染到页面上, MVVM 模式不需要用户收到操作 dom 元素,将数据绑定到 viewModel 层上,会自动将数据渲染到页面中,视图变化会通知 viewModel 层 更新数据。
viewModel 就是我们 MVVM 模式中的桥梁。

2.请说一下响应式数据的原理？

理解:

- 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。

原理:



```
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**/
  }
})

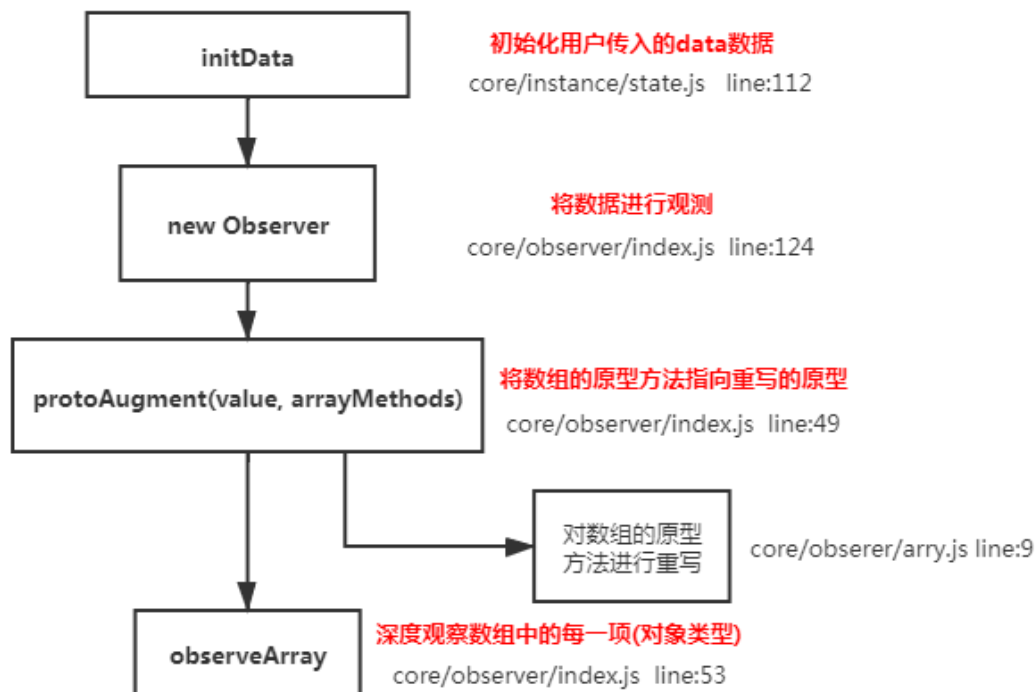
```

3. Vue 中是如何检测数组变化?

理解:

- 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。

原理:



```

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

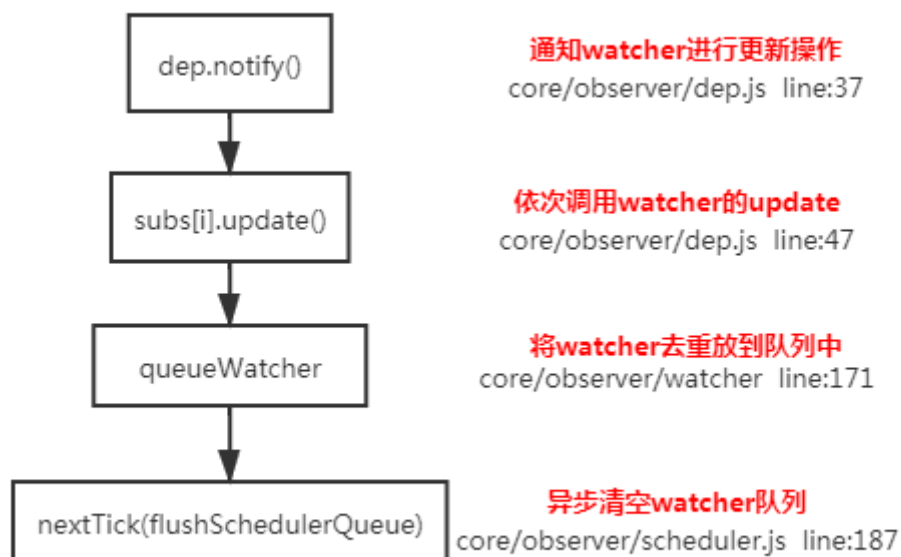
```

4.为何Vue采用异步渲染?

理解：

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!

原理:



```

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
  }
}
}

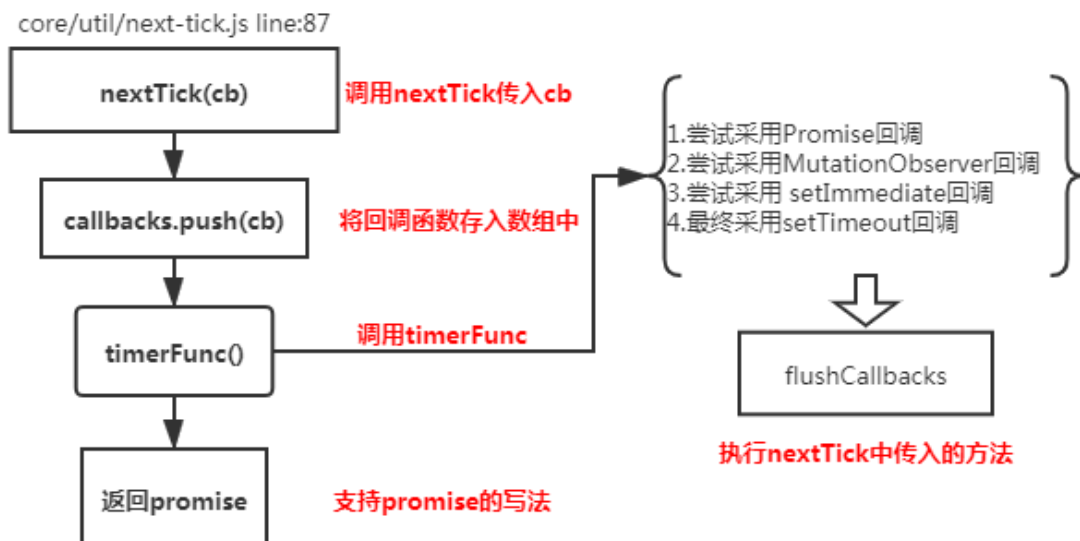
```

5.nextTick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

原理:



```
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()
  }
}

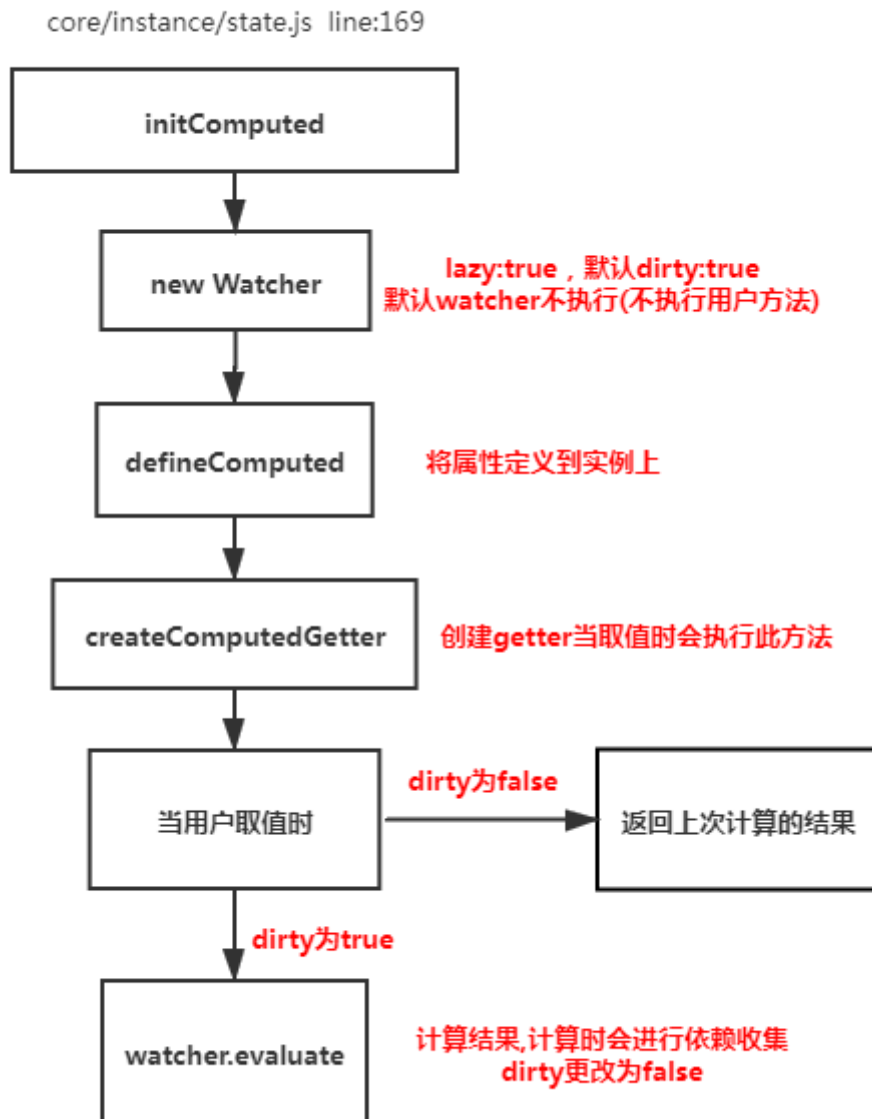
```

6. Vue 中 Computed 的特点

理解:

- 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

原理:



```
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
    }
  }
}
}

```

7. watch 中的 deep: true 是如何实现的

理解:

- 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

原理:

```

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)
  }
}
}

```

8. Vue 组件的生命周期

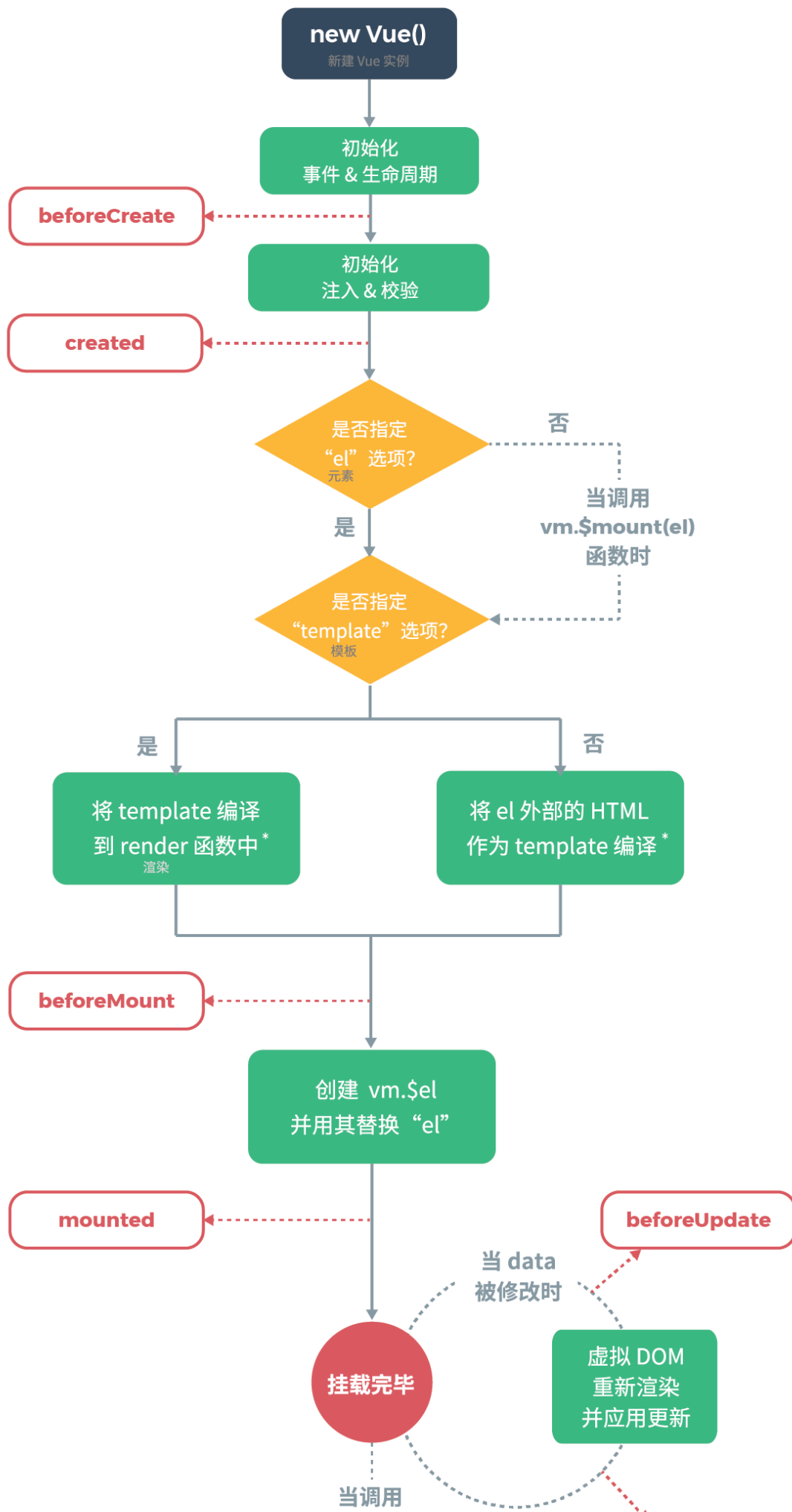
理解:

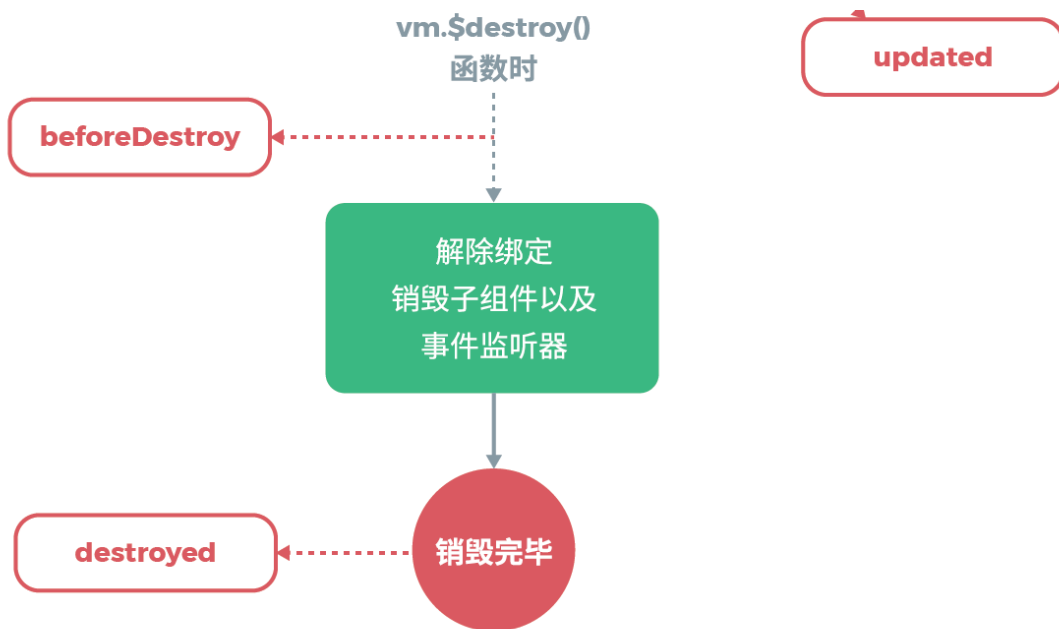
要掌握每个生命周期什么时候被调用

- `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





* 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

原理:

core/instance/index.js line:8

Vue

```
initMixin(Vue) 1.初始化_init方法
stateMixin(Vue) 2.初始化$set $delete $watch
eventsMixin(Vue) 3.初始化vue中的$on $emit事件
lifecycleMixin(Vue) 4.初始化_update方法
renderMixin(Vue) 5.初始化_render方法
```

this._init()

调用_init()之前 已经初始化了 vue的\$on/\$emit (vue的事件)

_init()

初始化vue的整个流程
core/instance/init.js line:16

```
initLifecycle(vm) // 1.初始化组件的父子关系
initEvents(vm) // 2.初始化组件事件
initRender(vm) // 3.初始化slot及createElement方法
callHook(vm, 'beforeCreate')
initInjections(vm) // 4.解析inject
initState(vm) // 5.初始化状态 响应式数据原理/method/watch
initProvide(vm) // 6.解析provide
callHook(vm, 'created')
```

beforeCreate
无法获取实例的数据

created中
可以获取数据

vm.\$mount

如果有el执行\$mount方法
platforms/web/entry-runtime-with-compiler line:18

判断是否有模板，并且将模板转化成render函数

mount.call()

调用runtime的mount方法
platforms/web/entry-runtime-with-compiler line:82

\$mount()

挂载组件
platforms/web/runtime/index.js line:42

mountComponent

进行组件的挂载
core/instance/lifecycle/index.js line:141

callHook(vm, 'beforeMount')

调用beforeMount方法
core/instance/lifecycle/index.js line:167

vm._update(vm._render())

初次渲染和更新会执行此方法
core/instance/lifecycle/index.js line:190

callHook(vm, 'mounted')

渲染完成后调用mounted钩子
core/instance/lifecycle/index.js line:210

当页面页面变化时
flushSchedulerQueue

触发beforeUpdate和updated
core/observer/scheduler.js line:71

当调用\$destroy()时

会触发beforeDestroy和destroyed
core/instance/lifecycle/index.js line:91

9. ajax 请求放在哪个生命周期中

理解:

- 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

10.何时需要使用 beforeDestroy

理解:

- 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`

11.Vue中模板编译原理

- 将 `template` 转化成 `render` 函数

```
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(`
```


12. Vue 中 v-if 和 v-show 的区别

理解:

- 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

原理:

```
const VueTemplateCompiler = require('vue-template-compiler');
let r1 = VueTemplateCompiler.compile(`<div v-if="true"><span v-for="i in 3">hello</span></div>`);
/**
with(this) {
  return (true) ? _c('div', _l((3), function (i) {
    return _c('span', [_v("hello")])
  }), 0) : _e()
}
*/
```

```
const VueTemplateCompiler = require('vue-template-compiler');
let r2 = VueTemplateCompiler.compile(`<div v-show="true"></div>`);
/**
with(this) {
  return _c('div', {
    directives: [{
      name: "show",
      rawName: "v-show",
      value: (true),
      expression: "true"
    }]
  })
}
*/

// v-show 操作的是样式 定义在platforms/web/runtime/directives/show.js
bind (el: any, { value }: VNodeDirective, vnode: VNodeWithData) {
  vnode = locateNode(vnode)
  const transition = vnode.data && vnode.data.transition
  const originalDisplay = el.__vOriginalDisplay =
    el.style.display === 'none' ? '' : el.style.display
  if (value && transition) {
    vnode.data.show = true
    enter(vnode, () => {
      el.style.display = originalDisplay
    })
  } else {
    el.style.display = value ? originalDisplay : 'none'
  }
}
```

13. 为什么 v-for 和 v-if 不能连用

理解:

```
const VueTemplateCompiler = require('vue-template-compiler');
let r1 = VueTemplateCompiler.compile(`

- 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



## 14.用 vnode 来描述一个 DOM 结构



- 虚拟节点就是用一个对象来描述真实的 dom 元素



```
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
 }
}
```



## 15.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  $O(n^3)$ ,vue 进行了优化: $O(n^3)$  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。



## 16.简述Vue 中 diff 算法原理



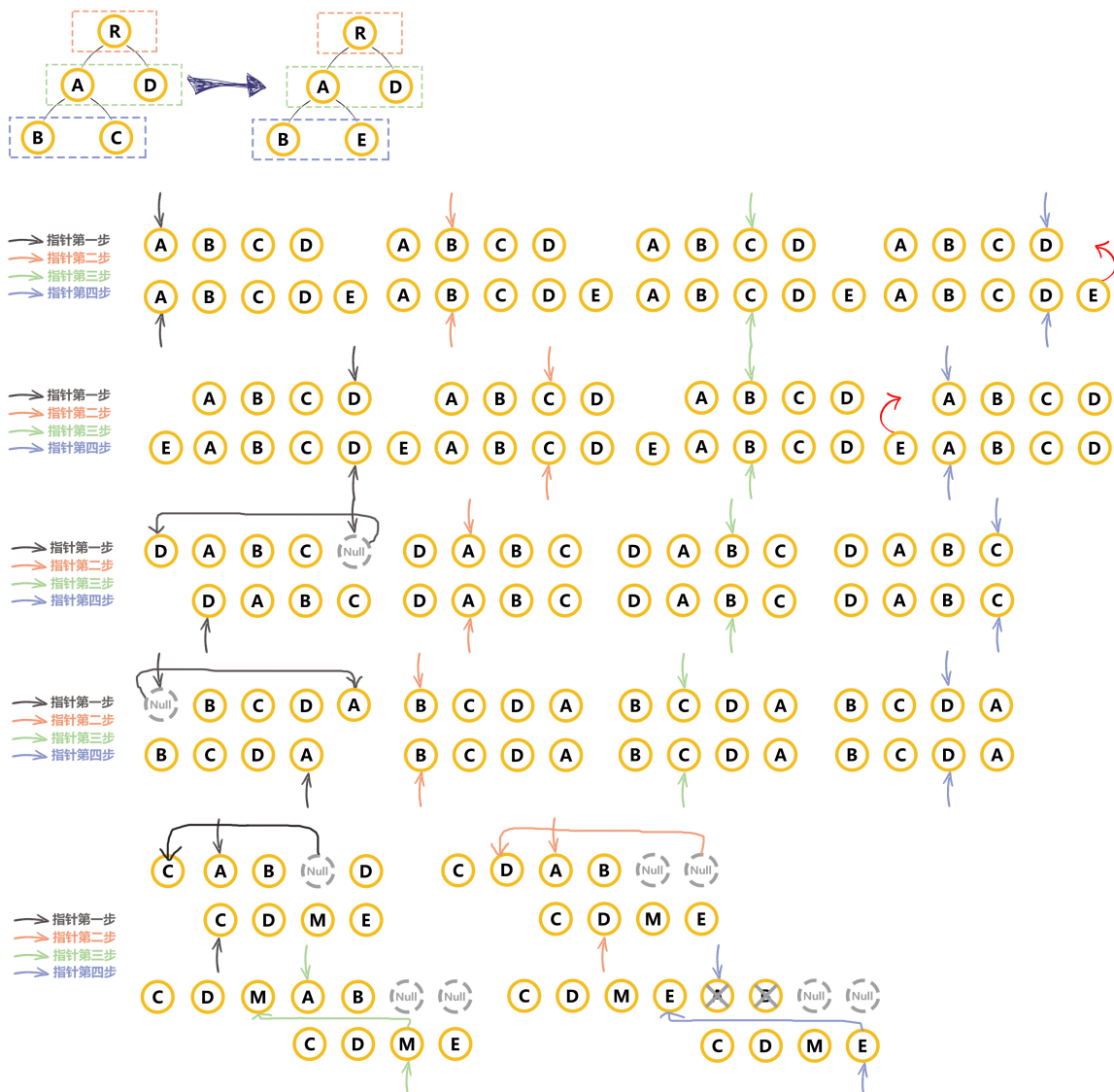
理解:



- 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



原理:

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)
    }
}

```



珠峰架构

微信号：zhufengjiagou

加微信可免费领取更多精彩视频

