《前端高手进阶》

朱德龙 前中兴软创主任工程师

— 拉勾教育出品 —

第18讲：你是怎么理解“组件”这个概念的

组件

拉勾教育
互联网人实战大学

不同框架、工具对组件的定义和实现各不相同

但可以用一句话来概括它们对组件的定义：

组件就是基于视图的模块

组件的**核心任务**就是将数据渲染到视图并监听用户在视图上的操作

视图

Vue 和 React 在编写组件视图的方式上有所不同

前者采用模板语言，更偏向于 HTML 语法

后者推荐使用语法糖 JSX，更偏向于 JavaScript 语法

但两者都是浏览器所无法直接识别的

所以都需要通过编译器转换成对应的可执行代码

1. 解析

```
while (html) {
 if (!lastTag || !isPlainTextElement(lastTag)) {
  let textEnd = html.indexOf('<')
  if (textEnd === 0) {
   if (comment.test(html)) {
    const commentEnd = html.indexOf('-->')
    if (commentEnd >= 0) {
     if (options.shouldKeepComment) {
      options.comment(html.substring(4, commentEnd),
index, index + commentEnd + 3)
     }
```

1. 解析

```
    advance(commentEnd + 3)
    continue
  }
}
if (conditionalComment.test(html)) {
  if (conditionalEnd >= 0) {
    advance(conditionalEnd + 2)
    continue
  }
}
}
```

1. 解析

```
      advance(conditionalEnd + 2)
      continue
    }
  }
 }
}
function advance (n) {
 index += n
 html = html.substring(n)
}
```

1. 解析

```
parseHTML(template, {
  start (tag, attrs, unary, start, end) {
    const ns = (currentParent && currentParent.ns) ||
platformGetTagNamespace(tag)
    if (isIE && ns === 'svg') {
      attrs = guardIESVGBug(attrs)
    }
    let element: ASTElement = createASTElement(tag, attrs, currentParent)
    if (ns) {
      element.ns = ns
    }
```

1. 解析

```
for (let i = 0; i < preTransforms.length; i++) {
  element = preTransforms[i](element, options) || element
}
if (!inVPre) {
  processPre(element)
  if (element.pre) {
    inVPre = true
  }
}
if (platformIsPreTag(element.tag)) {
  inPre = true
}
```

1. 解析

```
  }
  if (inVPre) {
    processRawAttrs(element)
  } else if (!element.processed) {
    processFor(element)
    processIf(element)
    processOnce(element)
  }
  if (!unary) {
    currentParent = element
    stack.push(element)
```

1. 解析

```
    processIf(element)
    processOnce(element)
  }
  if (!unary) {
    currentParent = element
    stack.push(element)
  } else {
    closeElement(element)
  }
}
})
```

1. 解析

```
Object
▶ attrs: [{…}]
▶ attrsList: [{…}]
▶ attrsMap: {id: "app"}
▶ children: (3) [{…}, {…}, {…}]
  end: 89
  parent: undefined
  plain: false
▶ rawAttrsMap: {id: {…}}
  start: 0
  tag: "div"
  type: 1
▶ __proto__: Object
```

2. 优化

```
function markStaticRoots (node: ASTNode, isInFor:
boolean) {
 if (node.type === 1) {
  if (node.static || node.once) {
   node.staticInFor = isInFor
  }
  if (node.static && node.children.length && !(
   node.children.length === 1 &&
   node.children[0].type === 3
  )) {
   node.staticRoot = true
```

2. 优化

```
    return
  }else {
    node.staticRoot = false
  }
  if (node.children) {
    for (let i = 0, l = node.children.length; i < l; i++) {
      markStaticRoots(node.children[i], isInFor
|| !!node.for)
    }
  }
  if (node.ifConditions) {
```

2. 优化

```
|| !!node.for)
    }
  }
  if (node.ifConditions) {
    for (let i = 1, l = node.ifConditions.length; i < l; i++) {
      markStaticRoots(node.ifConditions[i].block,
isInFor)
    }
  }
 }
}
```

3. 生成代码

```
// 视图模板
<div id="app">
  <h1>Hello {{text}}</h1>
  <span v-bind:id="message"></span>
</div>
//可执行的 js 代码
"with(this){return _c('div',{attrs:{"id":"app"}},[_c('h1',[_v("Hello
"+_s(text))]),_v(" "),_c('span',{attrs:{"id":message}})])}"
```

React

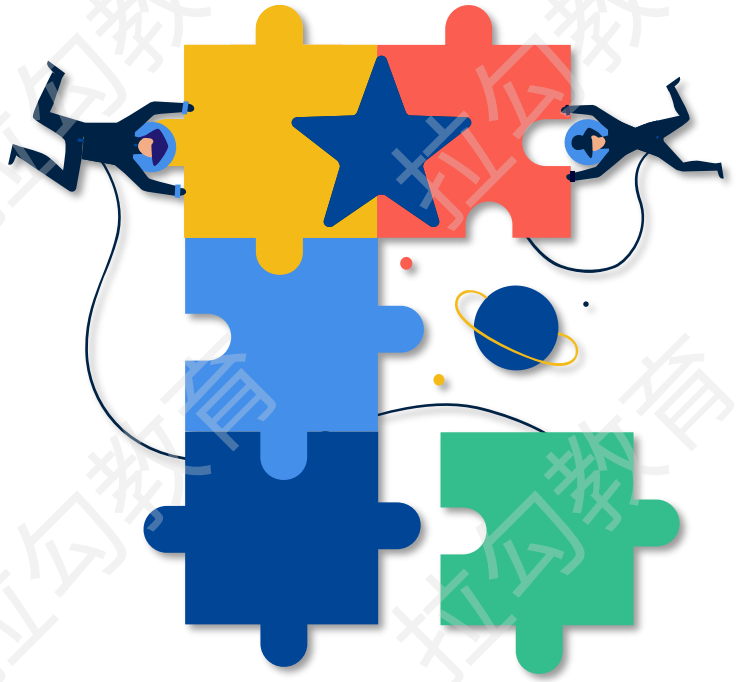
React 组件视图则使用 JS 的语法糖 jsx 来编写（不用 jsx 也可以编写组件）

这种语法糖其实就是混合了 HTML 和 JS 两种语言

浏览器也是无法直接识别的

所以用到了 babel 及其插件 babel-plugin-transform-react-jsx 对 jsx 进行预编译

编译步骤和之前提到的基本一致



延伸 1：虚拟 DOM 是用来提升性能的吗？

虽然 Vue 和 React 有着种种差异，但在某些地方达成了共识

比如都使用了虚拟 DOM 技术

对于使用过 React 或 Vue 的同学对虚拟 DOM 应该不陌生

其实就是 JavaScript 用来模拟真实 DOM 的数据对象

延伸 1：虚拟 DOM 是用来提升性能的吗？

- **优化性能**

  虚拟 DOM 提升了 DOM 操作的性能下限，降低了 DOM 操作的性能上限

- **跨平台**

  可根据不同的运行环境进行代码转换（比如浏览器、服务端、原生应用等）

  这使得它具有了跨平台的能力

数据模型

Vue

```
// 正确
Vue.component('item', {
  template: '<p>item:{{name}}</p>',
  // data 必须是函数
  data() {
    return { name: Math.random() }
  }
})
// 报错：The "data" option should be a function that returns a per-instance value in component definitions.
```

数据模型

Vue

```
  }
})
// 报错：The "data" option should be a function that returns a per-
instance value in component definitions.
Vue.component('item', {
  template: '<p>item:{{name}}</p>',
  data: {
    name: Math.random()
  }
})
```

数据模型

Vue

```
function initData (vm: Component) {
  let data = vm.$options.data
  data = vm._data = typeof data === 'function'
? getData(data, vm)
  : data || {}
  if (!isPlainObject(data)) {
    data = {}
    process.env.NODE_ENV !== 'production' && warn(
      'data functions should return an object:\n' +
```

数据模型

Vue

```
if (!isPlainObject(data)) {
  data = {}
  process.env.NODE_ENV !== 'production' && warn(
    'data functions should return an object:\n' +
    'https://vuejs.org/v2/guide/components.html#data-Must-Be-a-Function',
    vm
  )
}
```

数据模型

通过调用函数的方式确实可以保证每个组件实例拥有自己的数据

但如果 data 改成对象就一定不可以吗 ？

理论上通过深拷贝函数来创建数据对象副本也是完全可行的

数据模型

React

React 组件的数据模型 state，其值就是**对象类型**

但 React 并没有直接采用深拷贝的方式来实现

因为深拷贝操作性能**开销太大**

数据模型

React

```
// 创建对象
console.time('create')
var obj = {}
for(let i=0;i<100;i++) {
 obj[Math.random()] = Math.random()
}
console.timeEnd('create') // create:
0.288818359375ms
// 深拷贝
console.time('clone')
_.cloneDeep(obj)
console.timeEnd('clone') // clone: 0.637939453125ms
```

数据模型

React

```
let o = {val: 0}
let b = {val: 0}
class Child extends React.Component{
 constructor(){
  super()
  this.state = {
   o,
   b
  }
 }
```

数据模型

React

```
click(p){
 this.setState({
  [p]: {
   val: this.state[p].val+1
  }
 }, () =>{
  console.log('o:', this.state.o === o)
  console.log('b:', this.state.b === b)
 })
}
```

数据模型

React

```
render(){
 return (
  <div>
   <button onClick={this.click.bind(this, 'o')}>?é??o</button>
   <button onClick={this.click.bind(this, 'b')}>?é??b</button>
   <p>o.val: {this.state.o.val} </p>
   <p>b.val: {this.state.b.val} </p>
  </div>
 )
}
```

数据模型

React

```
  }
}
class App extends React.Component {
 render() {
  return (
    <div>
     <Child/>
    </div>
  )
 }
```

数据模型

React

```
    return (
      <div>
        <Child/>
      </div>
    )
  }
}
window.onload = function () {
  ReactDOM.render( <App/> , window.app)
}
```

渲染

Vue

Vue 采取的是响应式的视图更新方式

基于 Object.defineProperty() 函数

监听数据对象属性的变化，然后再更新到视图

渲染

Vue

```
export class Observer {
  value: any;
  dep: Dep;
  vmCount: number;
  constructor (value: any) {
    this.value = value
    this.dep = new Dep()
    this.vmCount = 0
    def(value, '__ob__', this)
    if (Array.isArray(value)) {
```

渲染

Vue

```
    if (hasProto) {
      protoAugment(value, arrayMethods)
    } else {
      copyAugment(value, arrayMethods, arrayKeys)
    }
    this.observeArray(value)
  } else {
    this.walk(value)
  }
}
```

渲染

Vue

```
walk (obj: Object) {
  const keys = Object.keys(obj)
  for (let i = 0; i < keys.length; i++) {
    defineReactive(obj, keys[i])
  }
}
observeArray (items: Array<any>) {
  for (let i = 0, l = items.length; i < l; i++) {
    observe(items[i])
  }
```

渲染

Vue

```
    for (let i = 0; i < keys.length; i++) {
      defineReactive(obj, keys[i])
    }
  }
  observeArray (items: Array<any>) {
    for (let i = 0, l = items.length; i < l; i++) {
      observe(items[i])
    }
  }
}
```

渲染

Vue

```
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()
 }
 if (getter && !setter) return
 if (setter) {
  setter.call(obj, newVal)
```

渲染

Vue

```
    customSetter()
  }
  if (getter && !setter) return
  if (setter) {
    setter.call(obj, newVal)
  } else {
    val = newVal
  }
  childOb = !shallow && observe(newVal)
  dep.notify()
}
```

Vue

```
export default class Dep {
  static target: ?Watcher;
  id: number;
  subs: Array<Watcher>;
  constructor () {
    this.id = uid++
    this.subs = []
  }
  addSub (sub: Watcher) {
    this.subs.push(sub)
  }
```

渲染

Vue

```
addSub (sub: Watcher) {
  this.subs.push(sub)
}
removeSub (sub: Watcher) {
  remove(this.subs, sub)
}
depend () {
  if (Dep.target) {
    Dep.target.addDep(this)
  }
}
```

渲染

Vue

```
}
notify () {
 const subs = this.subs.slice()
 if (process.env.NODE_ENV !== 'production' && !config.async) {
  subs.sort((a, b) => a.id - b.id)
 }
 for (let i = 0, l = subs.length; i < l; i++) {
  subs[i].update()
 }
}
}
```

渲染

Vue

```
export function queueWatcher (watcher: Watcher) {
 const id = watcher.id
 if (has[id] == null) {
  has[id] = true
  if (!flushing) {
   queue.push(watcher)
  } else {
   // if already flushing, splice the watcher based on its id
   // if already past its id, it will be run next immediately.
   let i = queue.length - 1
   while (i > index && queue[i].id > watcher.id) {
```

Vue

```
  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'
```

渲染

Vue

```
    if (process.env.NODE_ENV !== 'production'
&& !config.async) {
      flushSchedulerQueue()
      return
    }
    nextTick(flushSchedulerQueue)
  }
 }
}
```

渲染

Vue

```
if (typeof Promise !== 'undefined' && isNative(Promise)) {
  const p = Promise.resolve()
  timerFunc = () => {
    p.then(flushCallbacks)
    if (isIOS) setTimeout(noop)
  }
  isUsingMicroTask = true
} else if (!isIE && typeof MutationObserver !== 'undefined' && (
  isNative(MutationObserver) ||
  MutationObserver.toString() === '[object MutationObserverConstructor]'
)) {
```

Vue

```
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
```

Vue

```
  }
  isUsingMicroTask = true
} else if (typeof setImmediate !== 'undefined' && isNative(setImmediate)) {
  timerFunc = () => {
    setImmediate(flushCallbacks)
  }
} else {
  timerFunc = () => {
    setTimeout(flushCallbacks, 0)
  }
}
```

渲染

Vue

```
(function(){
 var obj = {id: 1}
 var array = []
 Object.defineProperty(o, 'obj', {
  enumerable: true,
  configurable: true,
  get: function() {
   return obj
  },
  set: function(val){
```

Vue

```
    set: function(val) {
      console.log('set object') // 不会执行
      obj = val
    }
  })

  Object.defineProperty(o, 'array', {
    enumerable: true,
    configurable: true,
    get: function() {
      return array
```

渲染

Vue

```
    },
    set: function(val) {
      console.log('set array') // 不会执行
      array = val
    }
  })
})()

o.obj.id = 2
console.log(o.obj); // {id: 2}
```

渲染

Vue

```
    console.log('set array') // 不会执行
    array = val
  }
})()

o.obj.id = 2
console.log(o.obj); // {id: 2}
o.array.push(1)
console.log(o.array); // [1]
```

渲染

Vue

```
template: '<p>item:{{o.name}}</p>',
data() {
  var data = {
    o: {
      name: 1
    }
  }
  return data
},
```

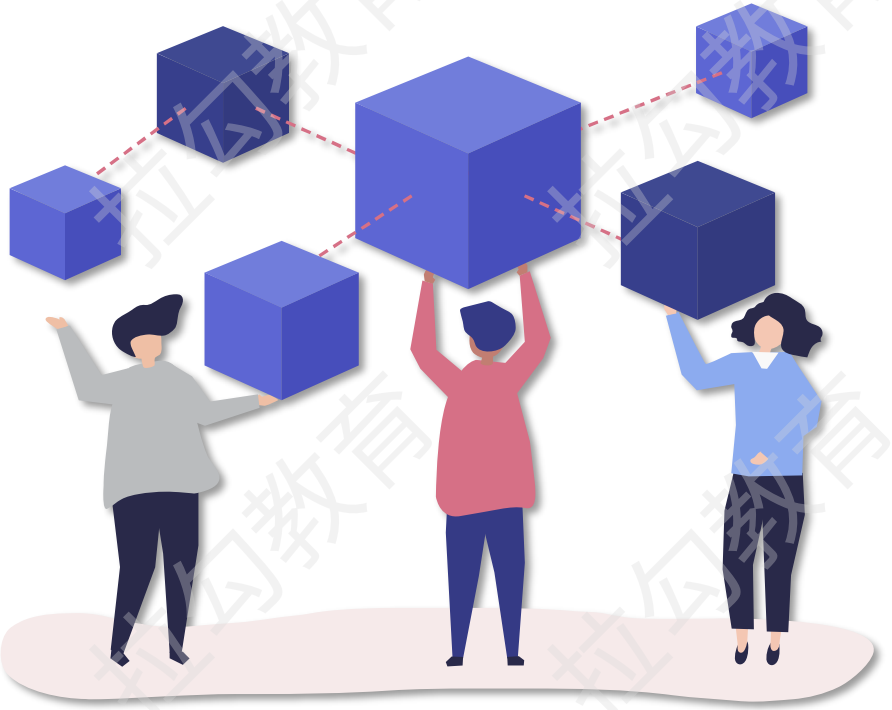
```
155
156   let childOb = !shallow && obse  e(val)   chil
157   Object.defineProperty(obj, key, {
158     enumerable: true,
154   }
155
156   let childOb = !shallow && obse  e(val)
157   Object.defineProperty(obj, key, {
158     enumerable: true,
```

"name"

"o"

渲染

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



Vue

```
const arrayProto = Array.prototype
export const arrayMethods = Object.create(arrayProto)
const methodsToPatch = [
  'push',
  'pop',
  'shift',
  'unshift',
  'splice',
  'sort',
  'reverse'
]
```

渲染

Vue

```
methodsToPatch.forEach(function (method) {
 const original = arrayProto[method]
 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
```

Vue

```
      inserted = args
      break
    case 'splice':
      inserted = args.slice(2)
      break
  }
  if (inserted) ob.observeArray(inserted)
  ob.dep.notify()
  return result
 })
})
```

渲染

Vue

```
export class Observer {
  value: any;
  dep: Dep;
  vmCount: number;
  constructor (value: any) {
    this.value = value
    this.dep = new Dep()
    this.vmCount = 0
    def(value, '__ob__', this)
    if (Array.isArray(value)) {
      if (hasProto) {
```

Vue

```
    protoAugment(value, arrayMethods)
  } else {
    copyAugment(value, arrayMethods, arrayKeys)
  }
  this.observeArray(value)
 } else {
  this.walk(value)
 }
}
walk (obj: Object) {
 const keys = Object.keys(obj)
```

Vue

```
    const keys = Object.keys(obj)
    for (let i = 0; i < keys.length; i++) {
      defineReactive(obj, keys[i])
    }
  }
  observeArray (items: Array<any>) {
    for (let i = 0, l = items.length; i < l; i++) {
      observe(items[i])
    }
  }
}
```

渲染

React

React 组件中的视图更新

并不是像 Vue 中那样自动响应的

而是需要**手动调用** setState() 函数来触发

React 为了提升组件更新时的性能

不仅将状态更新包装成任务放入了异步队列

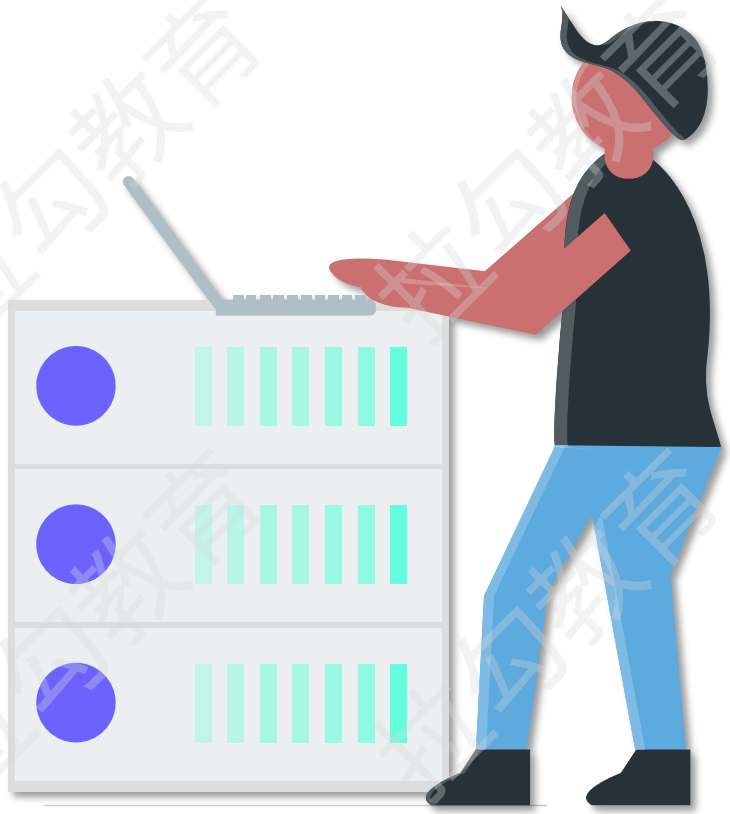
而且还使用了类似协程的方式来调度这些队列中的更新任务

任务的执行顺序会根据每个任务的优先级来进行调整

并且任务的执行过程中可能会被中断

但状态会被保存

直到合适的时候会再次读取状态并继续执行任务



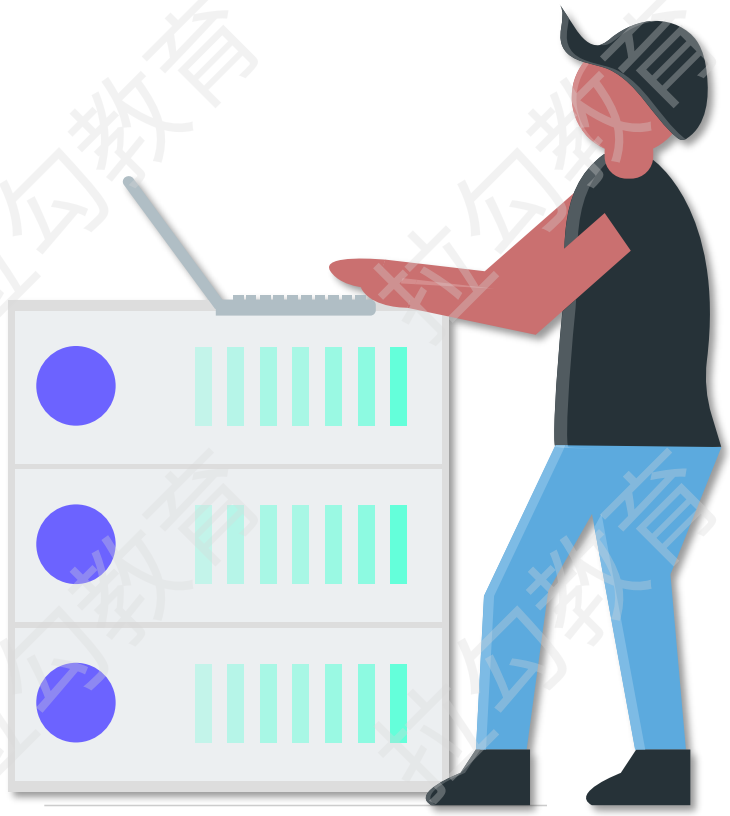
渲染

React

这种调度机制的具体表现就是：

在组件内部调用 setState() 来修改状态时将异步更新视图

而在原生 DOM 事件或异步操作中则是同步更新视图



总结

讲解了主流视图库 Vue 和 React 的组件实现机制

Vue 采用风格偏向 HTML 的模板语言

React 则采用了风格偏向 JavaScript 的 JSX 语法糖

Vue 通过函数来创建并返回数据对象

React 组件的状态对象则具有不可变性

Vue 通过监听数据对象属性实现响应式的数据绑定

React 则需要手动调用 setState() 函数才能触发更新

总结

拉勾教育
互联网人实战大学

你还知道哪些数据绑定的实现方式？

Next：第19讲：《把路由放在前端意味着什么？》

拉勾教育

—互联网人实战大学—

下载「拉勾教育App」
获取更多内容

L / A / G / O / U