

Vue 源码剖析-模板编译和组件化

模板编译

- 模板编译的主要目的是将模板 (template) 转换为渲染函数 (render)

```
1 <div>
2   <h1 @click="handler">title</h1>
3   <p>some content</p>
4 </div>
```

- 渲染函数 render

```
1 render (h) {
2   return h('div', [
3     h('h1', { on: { click: this.handler } }, 'title'),
4     h('p', 'some content')
5   ])
6 }
```

- 模板编译的作用
 - Vue 2.x 使用 VNode 描述视图以及各种交互，用户自己编写 VNode 比较复杂
 - 用户只需要编写类似 HTML 的代码 - Vue 模板，通过编译器将模板转换为返回 VNode 的 render 函数
 - .vue 文件会被 webpack 在构建的过程中转换成 render 函数

体验模板编译的结果

- 带编译器版本的 Vue.js 中，使用 template 或 el 的方式设置模板

```
1 <div id="app">
2   <h1>Vue<span>模板编译过程</span></h1>
3   <p>{{ msg }}</p>
4   <comp @myclick="handler"></comp>
5 </div>
6 <script src="../../dist/vue.js"></script>
7 <script>
8   vue.component('comp', {
9     template: '<div>I am a comp</div>'
10  })
11  const vm = new Vue({
12    el: '#app',
13    data: {
14      msg: 'Hello compiler'
15    },
16    methods: {
17      handler () {
18        console.log('test')
19      }
16    }
17  })
18  })
19  })
```

```

20     }
21   })
22   console.log(vm.$options.render)
23 </script>

```

- 编译后 render 输出的结果

```

1  (function anonymous() {
2    with (this) {
3      return _c(
4        "div",
5        { attrs: { id: "app" } },
6        [
7          _m(0),
8          _v(" "),
9          _c("p", [_v(_s(msg))]),
10         _v(" "),
11         _c("comp", { on: { myclick: handler } }),
12       ],
13       1
14     );
15   }
16 });

```

- `_c` 是 `createElement()` 方法, 定义的位置 `instance/render.js` 中
- 相关的渲染函数(开头的方法定义), 在 `instance/render-helps/index.js` 中

```

1  // instance/render-helps/index.js
2  target._v = createTextVNode
3  target._m = renderStatic
4
5  // core/vdom/vnode.js
6  export function createTextVNode (val: string | number) {
7    return new VNode(undefined, undefined, undefined, String(val))
8  }
9
10 // 在 instance/render-helps/render-static.js
11 export function renderStatic (
12   index: number,
13   isInFor: boolean
14 ): VNode | Array<VNode> {
15   const cached = this._staticTrees || (this._staticTrees = [])
16   let tree = cached[index]
17   // if has already-rendered static tree and not inside v-for,
18   // we can reuse the same tree.
19   if (tree && !isInFor) {
20     return tree
21   }
22   // otherwise, render a fresh tree.
23   tree = cached[index] = this.$options.staticRenderFns[index].call(
24     this._renderProxy,
25     null,
26     this // for render fns generated for functional component templates
27   )
28   markStatic(tree, `__static__${index}`, false)
29   return tree

```

- 把 template 转换成 render 的入口 src\platforms\web\entry-runtime-with-compiler.js

Vue Template Explorer

- [vue-template-explorer](#)
 - Vue 2.6 把模板编译成 render 函数的工具
- [vue-next-template-explorer](#)
 - Vue 3.0 beta 把模板编译成 render 函数的工具

模板编译过程

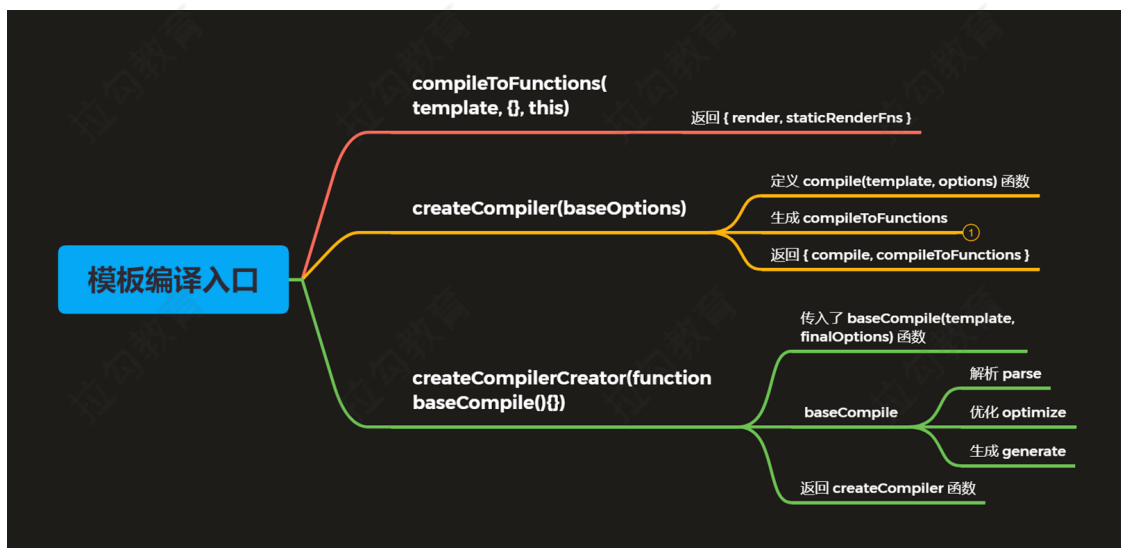
- 解析、优化、生成

编译的入口

- src\platforms\web\entry-runtime-with-compiler.js

```
1  vue.prototype.$mount = function (
2    .....
3    // 把 template 转换成 render 函数
4    const { render, staticRenderFns } = compileToFunctions(template, {
5      outputSourceRange: process.env.NODE_ENV !== 'production',
6      shouldDecodeNewlines,
7      shouldDecodeNewlinesForHref,
8      delimiters: options.delimiters,
9      comments: options.comments
10   }, this)
11   options.render = render
12   options.staticRenderFns = staticRenderFns
13   .....
14 )
```

- 调试 compileToFunctions() 执行过程，生成渲染函数的过程
 - compileToFunctions: src\compiler\to-function.js
 - compile(template, options): src\compiler\create-compiler.js
 - baseCompile(template.trim(), finalOptions): src\compiler\index.js



解析 - parse

- 解析器将模板解析为抽象语树 AST，只有将模板解析成 AST 后，才能基于它做优化或者生成代码字符串。
 - src\compiler\index.js

```

1  const ast = parse(template.trim(), options)
2
3  //src\compiler\parser\index.js
4  parse()

```

- 查看得到的 AST tree

[astexplorer](#)

- 结构化指令的处理
 - v-if 最终生成单元表达式

```

1  // src\compiler\parser\index.js
2  // structural directives
3  // 结构化的指令
4  // v-for
5  processFor(element)
6  processIf(element)
7  processOnce(element)
8
9  // src\compiler\codegen\index.js
10 export function genIf (
11   el: any,
12   state: CodegenState,
13   altGen?: Function,
14   altEmpty?: string
15 ): string {
16   el.ifProcessed = true // avoid recursion
17   return genIfConditions(el.ifConditions.slice(), state, altGen, altEmpty)
18 }
19 // 最终调用 genIfConditions 生成三元表达式

```

- v-if 最终编译的结果

```

1  f anonymous(
2  ) {
3    with(this){
4      return _c('div',{attrs:{"id":"app"}},[
5        _m(0),
6        _v(" "),
7        (msg)?_c('p',[_v(_s(msg))]):_e(),_v(" "),
8        _c('comp',{on:{"myclick":onMyClick}})
9      ],1)
10 }
11 }

```

v-if/v-for 结构化指令只能在编译阶段处理，如果我们要在 render 函数处理条件或循环只能使用 js 中的 if 和 for

```

1  Vue.component('comp', {
2    data: () {
3      return {
4        msg: 'my comp'
5      }
6    },
7    render (h) {
8      if (this.msg) {
9        return h('div', this.msg)
10     }
11     return h('div', 'bar')
12   }
13 })

```

优化 - optimize

- 优化抽象语法树，检测子节点中是否是纯静态节点
- 一旦检测到纯静态节点，例如：

hello整体是静态节点

永远不会更改的节点

- 提升为常量，重新渲染的时候不在重新创建节点
- 在 patch 的时候直接跳过静态子树

```

1  // src\compiler\index.js
2  if (options.optimize !== false) {
3    optimize(ast, options)
4  }
5
6  // src\compiler\optimizer.js
7  /**
8   * Goal of the optimizer: walk the generated template AST tree
9   * and detect sub-trees that are purely static, i.e. parts of
10  * the DOM that never needs to change.
11  *
12  * Once we detect these sub-trees, we can:
13  *

```

```

14  * 1. Hoist them into constants, so that we no longer need to
15  *   create fresh nodes for them on each re-render;
16  * 2. Completely skip them in the patching process.
17  */
18  export function optimize (root: ?ASTElement, options: CompilerOptions) {
19    if (!root) return
20    isStaticKey = genStaticKeysCached(options.staticKeys || '')
21    isPlatformReservedTag = options.isReservedTag || no
22    // first pass: mark all non-static nodes.
23    // 标记非静态节点
24    markStatic(root)
25    // second pass: mark static roots.
26    // 标记静态根节点
27    markStaticRoots(root, false)
28  }

```

生成 - generate

```

1  // src\compiler\index.js
2  const code = generate(ast, options)
3
4  // src\compiler\codegen\index.js
5  export function generate (
6    ast: ASTElement | void,
7    options: CompilerOptions
8  ): CodegenResult {
9    const state = new CodegenState(options)
10    const code = ast ? genElement(ast, state) : '_c("div")'
11    return {
12      render: `with(this){return ${code}}`,
13      staticRenderFns: state.staticRenderFns
14    }
15  }
16
17  // 把字符串转换成函数
18  // src\compiler\to-function.js
19  function createFunction (code, errors) {
20    try {
21      return new Function(code)
22    } catch (err) {
23      errors.push({ err, code })
24    }
25    return noop
26  }

```

组件化机制

- 组件化可以让我们方便的把页面拆分成多个可重用的组件
- 组件是独立的，系统内可重用，组件之间可以嵌套
- 有了组件可以像搭积木一样开发网页
- 下面我们将从源码的角度来分析 Vue 组件内部如何工作
 - 组件实例的创建过程是从上而下
 - 组件实例的挂载过程是从下而上

组件声明

- 复习全局组件的定义方式

```
1 Vue.component('comp', {
2   template: '<h1>hello</h1>'
3 })
```

- Vue.component() 入口
 - 创建组件的构造函数，挂载到 Vue 实例的 vm.options.component.componentName = Ctor

```
1 // src\core\global-api\index.js
2 // 注册 Vue.directive()、Vue.component()、Vue.filter()
3 initAssetRegisters(Vue)
4
5 // src\core\global-api\assets.js
6 if (type === 'component' && isPlainObject(definition)) {
7   definition.name = definition.name || id
8   definition = this.options._base.extend(definition)
9 }
10 .....
11 // 全局注册，存储资源并赋值
12 // this.options['components']['comp'] = Ctor
13 this.options[type + 's'][id] = definition
14
15
16 // src\core\global-api\index.js
17 // this is used to identify the "base" constructor to extend all plain-
18 // object
19 // components with in weex's multi-instance scenarios.
20 Vue.options._base = Vue
21
22 // src\core\global-api\extend.js
23 Vue.extend()
```

- 组件构造函数的创建

```
1 const Sub = function VueComponent (options) {
2   this._init(options)
3 }
4 Sub.prototype = Object.create(Super.prototype)
5 Sub.prototype.constructor = Sub
6 Sub.cid = cid++
7 Sub.options = mergeOptions(
8   Super.options,
9   extendOptions
10 )
11 Sub['super'] = Super
12
13 // For props and computed properties, we define the proxy getters on
14 // the Vue instances at extension time, on the extended prototype. This
15 // avoids Object.defineProperty calls for each instance created.
16 if (Sub.options.props) {
17   initProps(Sub)
```



```

18   }
19   if (Sub.options.computed) {
20     initComputed(Sub)
21   }
22
23   // allow further extension/mixin/plugin usage
24   Sub.extend = Super.extend
25   Sub.mixin = Super.mixin
26   Sub.use = Super.use
27
28   // create asset registers, so extended classes
29   // can have their private assets too.
30   ASSET_TYPES.forEach(function (type) {
31     Sub[type] = Super[type]
32   })
33   // enable recursive self-lookup
34   if (name) {
35     Sub.options.components[name] = Sub
36   }

```

- 调试 Vue.component() 调用的过程

```

1  <div id="app">
2  </div>
3  <script src="../../dist/vue.js"></script>
4  <script>
5    const Comp = Vue.component('comp', {
6      template: '<h2>I am a comp</h2>'
7    })
8    const vm = new Vue({
9      el: '#app',
10     render (h) {
11       return h(Comp)
12     }
13   })
14 </script>

```

组件创建和挂载

组件 VNode 的创建过程

- 创建根组件，首次 _render() 时，会得到整棵树的 VNode 结构
- 整体流程：new Vue() --> \$mount() --> vm._render() --> createElement() --> createComponent()
- 创建组件的 VNode，初始化组件的 hook 钩子函数

```

1  // 1. _createElement() 中调用 createComponent()
2  // src\core\vdom\create-element.js
3  else if ((!data || !data.pre) &&
4    isDef(Ctor = resolveAsset(context.$options, 'components', tag)))
5  {
6    // 查找自定义组件构造函数的声明
7    // 根据 Ctor 创建组件的 VNode
8    // component
9    vnode = createComponent(Ctor, data, context, children, tag)
10

```



```

11 // 2. createComponent() 中调用创建自定义组件对应的 VNode
12 // src\core\vdom\create-component.js
13 export function createComponent (
14   Ctor: Class<Component> | Function | Object | void,
15   data: ?VNodeData,
16   context: Component,
17   children: ?Array<VNode>,
18   tag?: string
19 ): VNode | Array<VNode> | void {
20   if (isUndef(Ctor)) {
21     return
22   }
23   .....
24   // install component management hooks onto the placeholder node
25   // 安装组件的钩子函数 init/prepatch/insert/destroy
26   // 初始化了组件的 data.hooks 中的钩子函数
27   installComponentHooks(data)
28
29   // return a placeholder vnode
30   const name = Ctor.options.name || tag
31   // 创建自定义组件的 VNode, 设置自定义组件的名字
32   // 记录this.componentOptions = componentOptions
33   const vnode = new VNode(
34     `vue-component-${Ctor.cid}${name ? `-${name}` : ''}`,
35     data, undefined, undefined, undefined, context,
36     { Ctor, propsData, listeners, tag, children },
37     asyncFactory
38   )
39   return vnode
40 }
41
42
43 // 3. installComponentHooks() 初始化组件的 data.hook
44 function installComponentHooks (data: VNodeData) {
45   const hooks = data.hook || (data.hook = {})
46   // 用户可以传递自定义钩子函数
47   // 把用户传入的自定义钩子函数和 componentVNodeHooks 中预定义的钩子函数合并
48   for (let i = 0; i < hooksToMerge.length; i++) {
49     const key = hooksToMerge[i]
50     const existing = hooks[key]
51     const toMerge = componentVNodeHooks[key]
52     if (existing !== toMerge && !(existing && existing._merged)) {
53       hooks[key] = existing ? mergeHook(toMerge, existing) : toMerge
54     }
55   }
56 }
57
58 // 4. 钩子函数定义的位置 (init()) 钩子中创建组件的实例)
59 // inline hooks to be invoked on component VNodes during patch
60 const componentVNodeHooks = {
61   init (vnode: VNodeWithData, hydrating: boolean): ?boolean {
62     if (
63       vnode.componentInstance &&
64       !vnode.componentInstance._isDestroyed &&
65       vnode.data.keepAlive
66     ) {
67       // kept-alive components, treat as a patch
68       const mountedNode: any = vnode // work around flow

```

```

69     componentVNodeHooks.prepatch(mountedNode, mountedNode)
70   } else {
71     // 创建组件实例挂载到 vnode.componentInstance
72     const child = vnode.componentInstance =
createComponentInstanceForVnode(
73       vnode,
74       activeInstance
75     )
76     // 调用组件对象的 $mount(), 把组件挂载到页面
77     child.$mount(hydrating ? vnode.elm : undefined, hydrating)
78   }
79 },
80
81 prepatch (oldVnode: MountedComponentVNode, vnode: MountedComponentVNode)
82 {
83   .....
84 },
85 insert (vnode: MountedComponentVNode) {
86   .....
87 },
88
89 destroy (vnode: MountedComponentVNode) {
90   .....
91 }
92 }
93
94 //5 .创建组件实例的位置, 由自定义组件的 init() 钩子方法调用
95 export function createComponentInstanceForVnode (
96   vnode: any, // we know it's MountedComponentVNode but flow doesn't
97   parent: any, // activeInstance in lifecycle state
98 ): Component {
99   const options: InternalComponentOptions = {
100     _isComponent: true,
101     _parentVnode: vnode,
102     parent
103   }
104   // check inline-template render functions
105   const inlineTemplate = vnode.data.inlineTemplate
106   if (isDef(inlineTemplate)) {
107     options.render = inlineTemplate.render
108     options.staticRenderFns = inlineTemplate.staticRenderFns
109   }
110   // 创建组件实例
111   return new vnode.componentOptions.Ctor(options)
112 }

```

- 调试执行过程

组件实例的创建和挂载过程

- Vue._update() --> patch() --> createElm() --> createComponent()

```

1 // src\core\vdom\patch.js
2 // 1. 创建组件实例, 挂载到真实 DOM
3 function createComponent (vnode, insertedVnodeQueue, parentElm, refElm) {
4   let i = vnode.data

```

```

5   if (isDef(i)) {
6       const isReactivated = isDef(vnode.componentInstance) && i.keepAlive
7       if (isDef(i = i.hook) && isDef(i = i.init)) {
8           // 调用 init() 方法, 创建和挂载组件实例
9           // init() 的过程中创建好了组件的真实 DOM, 挂载到了 vnode.elm 上
10          i(vnode, false /* hydrating */)
11      }
12      // after calling the init hook, if the vnode is a child component
13      // it should've created a child instance and mounted it. the child
14      // component also has set the placeholder vnode's elm.
15      // in that case we can just return the element and be done.
16      if (isDef(vnode.componentInstance)) {
17          // 调用钩子函数 (VNode的钩子函数初始化属性/事件/样式等, 组件的钩子函数)
18          initComponent(vnode, insertedVnodeQueue)
19          // 把组件对应的 DOM 插入到父元素中
20          insert(parentElm, vnode.elm, refElm)
21          if (isTrue(isReactivated)) {
22              reactivateComponent(vnode, insertedVnodeQueue, parentElm, refElm)
23          }
24          return true
25      }
26  }
27  }
28
29  // 2. 调用钩子函数, 设置局部作用于样式
30  function initComponent (vnode, insertedVnodeQueue) {
31      if (isDef(vnode.data.pendingInsert)) {
32          insertedVnodeQueue.push.apply(insertedVnodeQueue,
33              vnode.data.pendingInsert)
34          vnode.data.pendingInsert = null
35      }
36      vnode.elm = vnode.componentInstance.$el
37      if (isPatchable(vnode)) {
38          // 调用钩子函数
39          invokeCreateHooks(vnode, insertedVnodeQueue)
40          // 设置局部作用于样式
41          setScope(vnode)
42      } else {
43          // empty component root.
44          // skip all element-related modules except for ref (#3455)
45          registerRef(vnode)
46          // make sure to invoke the insert hook
47          insertedVnodeQueue.push(vnode)
48      }
49  }
50
51  // 3. 调用钩子函数
52  function invokeCreateHooks (vnode, insertedVnodeQueue) {
53      // 调用 VNode 的钩子函数, 初始化属性/样式/事件等
54      for (let i = 0; i < cbs.create.length; ++i) {
55          cbs.create[i](emptyNode, vnode)
56      }
57      i = vnode.data.hook // Reuse variable
58      // 调用组件的钩子函数
59      if (isDef(i)) {
60          if (isDef(i.create)) i.create(emptyNode, vnode)
61          if (isDef(i.insert)) insertedVnodeQueue.push(vnode)
62      }
63  }

```



扫码联系老师

技能评估、福利资料、课程优惠

Made with ❤️ by LagouFed