32 | 编译原理（上）：手写一个迷你Vue 3 Compiler的入门原理

2022-01-03 大圣

《玩转Vue 3全家桶》 课程介绍 >

讲述：大圣

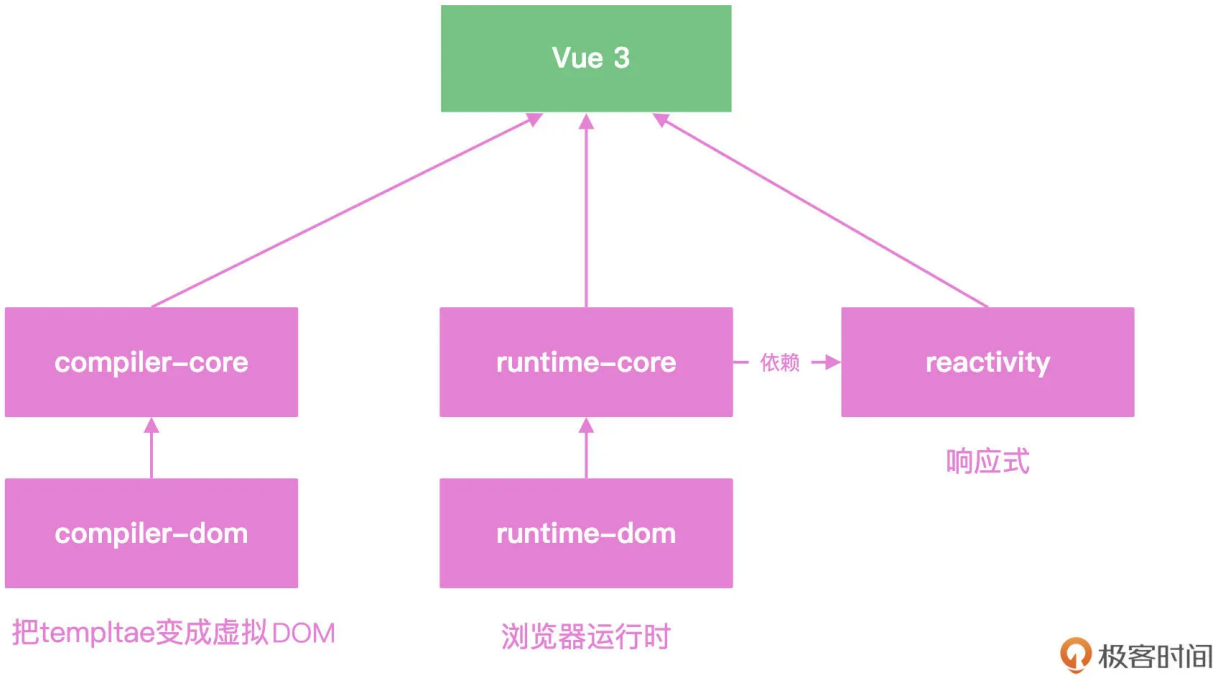
时长 07:48 大小 7.16M

你好，我是大圣。

前面我们用了四讲，学习了 Vue 在浏览器中是如何执行的，你可以参考上一讲结尾的 Vue 执行全景图来回顾一下。在 Vue 中，组件都是以虚拟 DOM 的形式存在，加载完毕之后注册 effect 函数。这样组件内部的数据变化之后，用 Vue 的响应式机制做到了通知组件更新，内部则使用 patch 函数实现了虚拟 DOM 的更新，中间我们也学习了位运算、最长递增子序列等算法。

这时候你肯定还有一个疑问，那就是虚拟 DOM 是从哪来的？我们明明写的是 templa 和 JSX，这也是吃透 Vue 源码最后一个难点：Vue 中的 Compiler。

下图就是 Vue 核心模块依赖关系图，reactivity 和 runtime 我们已经剖析完毕，迷你版本的代码你可以在GitHub中看到。今天开始我将用三讲的内容，给你详细讲解一下 Vue 在编译的过程中做了什么。



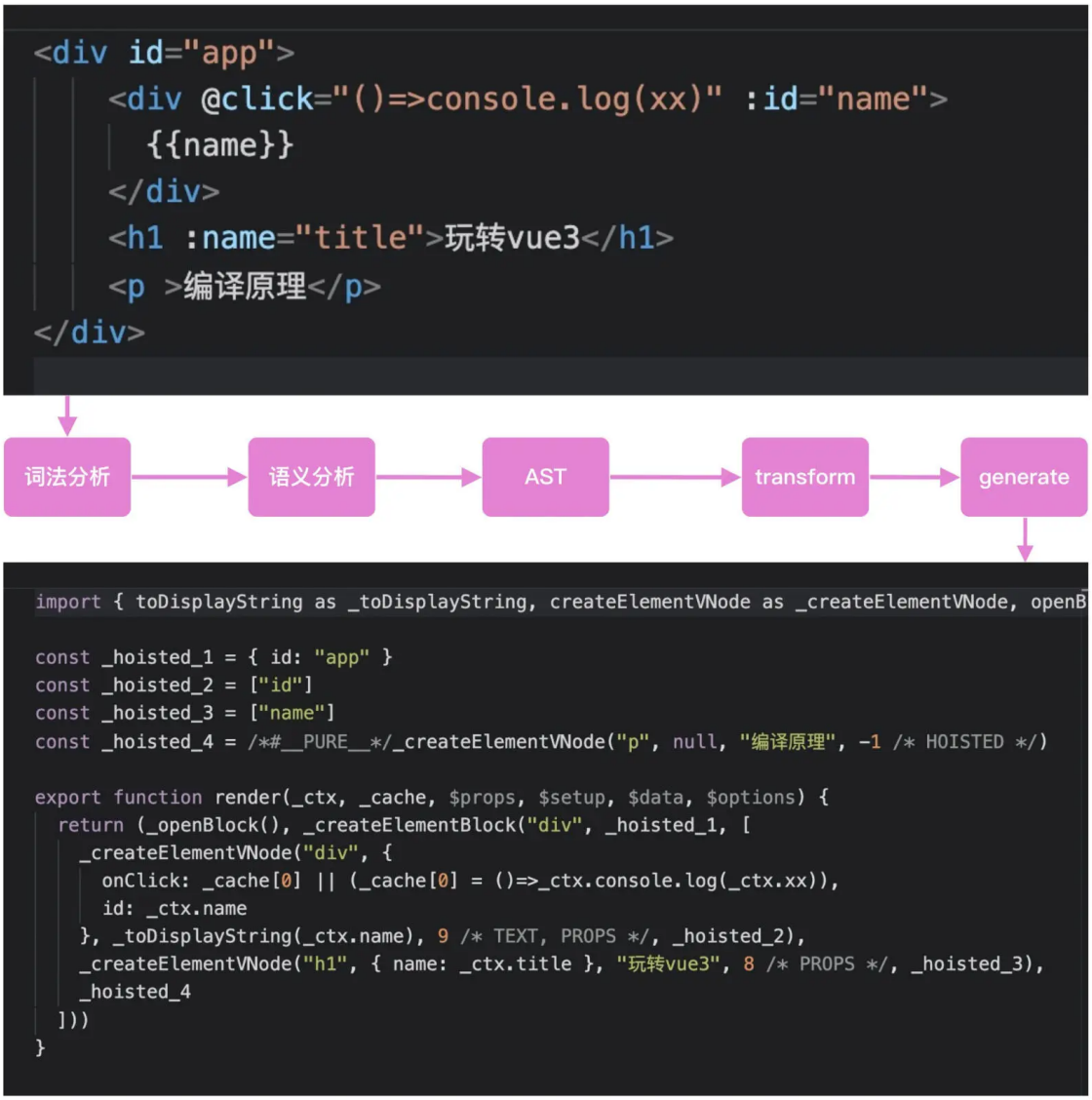
编译原理也属于计算机中的一个重要学科，Vue 的 compiler 是在 Vue 场景下的实现，目的就是实现 template 到 render 函数的转变。

我们第一步需要先掌握编译原理的基本概念。Vue 官方提供了模板编译的在线演示。下图左侧代码是我们写的 template，右侧代码就是 compiler 模块解析城的 render 函数，我们今天的任务就是能够实现一个迷你的 compiler。

整体流程

上述转化的过程可以分为下面的示意图几步来实现。

首先，代码会被解析成一个对象，这个对象有点像虚拟 DOM 的概念，用来描述 template 的代码关系，这个对象就是抽象语法树（简称 AST，后面我们细讲）。然后通过 transform 模块对代码进行优化，比如识别 Vue 中的语法，静态标记、最后通过 generate 模块生成最终的 render 函数。



理清了流程，我们动手完成具体代码实现。用下面的代码就能实现上述的流程图里的内容。其中 parse 函数负责生成抽象语法树 AST，transform 函数负责语义转换，generate 函数负责最终的代码生成。

复制代码

```
1
2 function compiler(template) {
3   const ast = parse(template);
4   transform(ast)
```

```
  const code = generate(ast)
  return code
}

let template = `<div id="app">
  <div @click="()=>console.log(xx)" :id="name">{{name}}</div>
  <h1 :name="title">玩转vue3</h1>
  <p >编译原理</p>
</div>
`

const renderFunction = compiler(template)
console.log(renderFunction)
```

我们先来看下 parse 函数如何实现。template 转成 render 函数是两种语法的转换，这种代码转换的需求其实计算机的世界中非常常见。比如我们常用的 Babel，就是把 ES6 的语法转成低版本浏览器可以执行的代码。

tokenizer 的迷你实现

首先，我们要对 template 进行词法分析，把模板中的

，

下面的代码就是 tokenizer 的迷你实现。我们使用 tokens 数组存储解析的结果，然后对模板字符串进行循环，在 template 中，< > / 和空格都是关键的分隔符，如果碰见 < 字符，我们需要判断下一个字符的状态。如果是字符串我们就标记 tagstart；如果是 /，我们就知道是结束标签，标记为 tagend，最终通过 push 方法把分割之后的 token 存储在数组 tokens 中返回。

复制代码

```
function tokenizer(input) {
  let tokens = []
  let type = ''
  let val = ''
  // 粗暴循环
  for (let i = 0; i < input.length; i++) {
    let ch = input[i]
    if (ch === '<') {
      push()
      if (input[i + 1] === '/') {
        type = 'tagend'
      } else {
```

```
        type = 'tagstart'
      }
    } if (ch === '>') {
      if(input[i-1]=='='){
        //箭头函数
      }else{
        push()
        type = "text"
        continue
      }
    } else if (/[\s]/.test(ch)) { // 碰见空格截断一下
      push()
      type = 'props'
      continue
    }
    val += ch
  }
  return tokens

  function push() {
    if (val) {
      if (type === "tagstart") val = val.slice(1) // <div => div
      if (type === "tagend") val = val.slice(2)   //  </div  => div
      tokens.push({
        type,
        val
      })
      val = ''
    }
  }
}
```

实现了上面的代码，我们就得到了解析之后的 token 数组。

生成抽象语法树

下面的数组中，我们分别用 tagstart、props tagend 和 text 标记，用它们标记了全部内容。然后下一步我们需要把这个数组按照标签的嵌套关系转换成树形结构，这样才能完整地描述 template 标签的关系。

复制代码

```
[
  { type: 'tagstart', val: 'div' },
  { type: 'props', val: 'id="app"' },
  { type: 'tagstart', val: 'div' },
  { type: 'props', val: '@click="()=console.log(xx)"' },
  { type: 'props', val: ':id="name"' },
```

```
  { type: 'text', val: '{{name}}' },
  { type: 'tagend', val: 'div' },
  { type: 'tagstart', val: 'h1' },
  { type: 'props', val: ':name="title"' },
  { type: 'text', val: '玩转vue3' },
  { type: 'tagend', val: 'h1' },
  { type: 'tagstart', val: 'p' },
  { type: 'text', val: '编译原理' },
  { type: 'tagend', val: 'p' },
  { type: 'tagend', val: 'div' }
```

然后我们分析 token 数组，看看它是如何转化成一个体现语法规则的树形结构的。

就像我们用虚拟 DOM 描述页面 DOM 结构一样，我们使用树形结构描述 template 的语法，这个树我们称之为抽象语法树，简称 AST。

下面的代码中我们用 parse 函数实现 AST 的解析。过程是这样的，首先我们使用一个 AST 对象作为根节点。然后通过 walk 函数遍历整个 tokens 数组，根据 token 的类型不同，生成不同的 node 对象。最后根据 tagend 的状态来决定 walk 的递归逻辑，最终实现了整棵树的构建。

复制代码

```
function parse(template) {
  const tokens = tokenizer(template)
  let cur = 0
  let ast = {
    type: 'root',
    props:[],
    children: []
  }
  while (cur < tokens.length) {
    ast.children.push(walk())
  }
  return ast

  function walk() {
    let token = tokens[cur]
    if (token.type == 'tagstart') {
      let node = {
        type: 'element',
        tag: token.val,
        props: [],
        children: []
      }
      token = tokens[++cur]
```

```
      while (token.type !== 'tagend') {
        if (token.type == 'props') {
          node.props.push(walk())
        } else {
          node.children.push(walk())
        }
        token = tokens[cur]
      }
      cur++
      return node
    }
    if (token.type === 'tagend') {
      cur++
      // return token
    }
    if (token.type == "text") {
      cur++
      return token
    }
    if (token.type === "props") {
      cur++
      const [key, val] = token.val.replace('=','~').split('~')
      return {
        key,
        val
      }
    }
  }
}
```

上面的代码会生成抽象语法树 AST，这个树的结构如下面代码所示，通过 type 和 children 描述整个 template 的结构。

复制代码

```
{
  "type": "root",
  "props": [],
  "children": [
    {
      "type": "element",
      "tag": "div",
      "props": [
        {
          "key": "id",
          "val": "\"app\""
        }
      ],
      "children": [
```

```
        {
          "type": "element",
          "tag": "div",
          "props": [
            {
              "key": "@click",
              "val": "\"()"
            },
            {
              "key": ":id",
              "val": "\"name\""
            }
          ],
          "children": [
            {
              "type": "text",
              "val": "{{name}}"
            }
          ]
        },
        {
          "type": "element",
          "tag": "h1",
          "props": [
            {
              "key": ":name",
              "val": "\"title\""
            }
          ],
          "children": [
            {
              "type": "text",
              "val": "玩转vue3"
            }
          ]
        },
        {
          "type": "element",
          "tag": "p",
          "props": [],
          "children": [
            {
              "type": "text",
              "val": "编译原理"
            }
          ]
        }
      ]
    }
  ]
}
```

语义分析和优化

有了抽象语法树之后，我们还要进行语义的分析和优化，也就是说，我们要在这个阶段理解语句要做的事。咱们结合例子来理解会更容易。

在 template 这个场景下，两个大括号包裹的字符串就是变量，@click 就是事件监听。

下面的代码中我们使用 transform 函数实现这个功能，这一步主要是理解 template 中 Vue 的语法，并且为最后生成的代码做准备。我们使用 context 对象存储 AST 所需要的上下文，如果我们用到了变量{{}}，就需要引入 toDisplayString 函数，上下文中的 helpers 存储的就是我们用到的工具函数。

复制代码

```
function transform(ast) {
  // 优化一下ast
  let context = {
    // import { toDisplayString , createVNode , openBlock , createBlock } from
    helpers:new Set(['openBlock','createVnode']), // 用到的工具函数
  }
  traverse(ast, context)
  ast.helpers = context.helpers
}
```

然后我们使用 traverse 函数递归整个 AST，去优化 AST 的结构，并且在这一步实现简单的静态标记。

当节点标记为 element 的时候，我们递归调用整个 AST，内部挨个遍历 AST 所有的属性，我们默认使用 ast.flag 标记节点的动态状态。如果属性是 @开头的，我们就认为它是 Vue 中的事件绑定，使用 arg.flag|= PatchFlags.EVENT 标记当前节点的事件是动态的，需要计算 diff，这部分位运算的知识点我们在上一讲已经学习过了。

然后冒号开头的就是动态的属性传递，并且把 class 和 style 标记了不同的 flag。如果都没有命中的话，就使用 static:true，标记当前节点位是静态节点。

复制代码

```
function traverse(ast, context){
  switch(ast.type){
    case "root":
```

```
      context.helpers.add('createBlock')
      // log(ast)
    case "element":
      ast.children.forEach(node=>{
        traverse(node,context)
      })
      ast.flag = 0
      ast.props = ast.props.map(prop=>{
        const {key,val} = prop
        if(key[0]=='@'){
          ast.flag |= PatchFlags.EVENT // 标记event需要更新
          return {
            key:'on'+key[1].toUpperCase()+key.slice(2),
            val
          }
        }
        if(key[0]==':'){
          const k = key.slice(1)
          if(k=="class"){
            ast.flag |= PatchFlags.CLASS // 标记class需要更新

          }else if(k=='style'){
            ast.flag |= PatchFlags.STYLE // 标记style需要更新
          }else{
            ast.flag |= PatchFlags.PROPS // 标记props需要更新
          }
          return{
            key:key.slice(1),
            val
          }
        }
        if(key.startsWith('v-')){
          // pass such as v-model
        }
        //标记static是true 静态节点
        return {...prop,static:true}
      })
      break
    case "text":
      // trnsformText
      let re = /\{\{(.*)\}\}/g
      if(re.test(ast.val)){
        //有{{
          ast.flag |= PatchFlags.TEXT // 标记props需要更新
          context.helpers.add('toDisplayString')
          ast.val = ast.val.replace(/\{\{(.*)\}\}/g,function(s0,s1){
            return s1
          })
      }else{
        ast.static = true
      }
  }
```

```
56 }
57
```

经过上面的代码标记优化之后，项目在数据更新之后，浏览器计算虚拟 dom diff 运算的时候，就会执行类似下面的代码逻辑。

我们通过在 compiler 阶段的标记，让 template 产出的虚拟 DOM 有了更精确的状态，可以越过大部分的虚拟 DOM 的 diff 计算，极大提高 Vue 的运行时效率，这个思想我们日常开发中也可以借鉴学习。

复制代码

```
if(vnode.static){
  return
}
if(vnode.flag & patchFlag.CLASS){
  遍历class 计算diff
}else if(vnode.flag & patchFlag.STYLE){
  计算style的diff
}else if(vnode.flag & patchFlag.TEXT){
  计算文本的diff
}
```

接下来，我们基于优化之后的 AST 生成目标代码，也就是 generate 函数要做的事：遍历整个 AST，拼接成最后要执行的函数字符串。

下面的代码中，我们首先把 helpers 拼接成 import 语句，并且使用 walk 函数遍历整个 AST，在遍历的过程中收集 helper 集合的依赖。最后，在 createVnode 的最后一个参数带上 ast.flag 进行状态的标记。

复制代码

```
function generate(ast) {
  const {helpers} = ast

  let code = `
import {${[...helpers].map(v=>v+' as _'+v).join(',')}} from 'vue'\n
export function render(_ctx, _cache, $props){
  return(_openBlock(), ${ast.children.map(node=>walk(node))})}`

  function walk(node){
    switch(node.type){
      case 'element':
```

```
        let {flag} = node // 编译的标记
        let props = '{'+node.props.reduce((ret,p)=>{
          if(flag.props){
            //动态属性
            ret.push(p.key +':_ctx.'+p.val.replace(/['"]/g,'') )
          }else{
            ret.push(p.key +':'+p.val )
          }

          return ret
        },[]).join(',')+'}'
        return `_createVnode("${node.tag}",${props}),[
          ${node.children.map(n=>walk(n))}
        ],${JSON.stringify(flag)}`
        break
      case 'text':
        if(node.static){
          return '"'+node.val+'"'
        }else{
          return `_toDisplayString(_ctx.${node.val})`
        }
        break
    }
  }
  return code
}
```

最终实现效果

最后我们执行一下代码，看下效果输出的代码。可以看到，它已经和 Vue 输出的代码很接近了，到此为止，我们也实现了一个非常迷你的 Vue compiler，这个产出的 render 函数最终会和组件的 setup 函数一起组成运行时的组件对象。

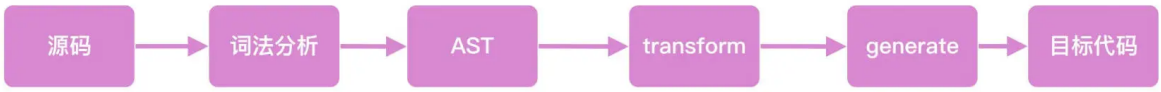
复制代码

```
function compiler(template) {
  const ast = parse(template);
  transform(ast)

  const code = generate(ast)
  return code
}

let template = `<div id="app">
  <div @click="()=>console.log(xx)" :id="name">{{name}}</div>
  <h1 :name="title">玩转vue3</h1>
  <p >编译原理</p>
</div>
`
```

```
15
16 const renderFunction = compiler(template)
17 console.log(renderFunction)
18
19 // 下面是输出结果
20 import { openBlock as _openBlock, createVnode as _createVnode, createBlock as
21
22 export function render(_ctx, _cache, $props) {
23   return (_openBlock(), _createVnode("div", { id: "app" }), [
24     _createVnode("div", { onClick: "()=>console.log(xx)", id: "name" }), [
25       _toDisplayString(_ctx.name)
26     ], 24, _createVnode("h1", { name: "title" }), [
27       "玩转vue3"
28     ], 8, _createVnode("p", {}), [
29       "编译原理"
30     ], 0
31   ], 0)
32 }
33
```

总结

我们总结一下今天所学的内容。今天，我带你手写了一个非常迷你的 Vue compiler，这也是我们学习框架源码的时候一个比较正确的思路：在去看实际的源码之前，先通过迷你版本的实现，熟悉整个 Vue compiler 工作的主体流程。



通过这个迷你的 compiler，我们学习了编译原理的入门知识：包括 parser 的实现、AST 是什么，AST 的语义化优化和代码生成 generate 模块，这给我们下一讲弄清楚 Vue 的 compiler 的核心逻辑打下了良好的理论基础。

我想提醒你注意一个优化方法，我们实现的迷你 compiler 也实现了属性的静态标记，通过在编译期间的标记方式，让虚拟 DOM 在运行时有更多的状态，从而能够精确地控制更新。这种编译时的优化也能够对我们项目开发有很多指引作用，我会在剖析完 Vue 的 compiler 之后，在第 34 讲那里跟你分享一下实战中如何使用编译优化的思想。

思考题

最后留一个思考题吧，Vue 的 compiler 输出的代码会有几个 hoisted 开头的变量，这几个变量有什么用处呢？欢迎在评论区分享你的答案，也欢迎你把这一讲分享给你的同事和朋友们，我们下一讲再见！

上一篇 31 | 虚拟DOM（下）：想看懂虚拟DOM算法，先刷个算法题

精选留言

> 由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。

写留言