



下载APP



34 | 编译原理（下）：编译原理给我们带来了什么？

2022-01-07 大圣

《玩转Vue 3全家桶》

课程介绍 >

**讲述：大圣**

时长 07:46 大小 7.12M



你好，我是大圣。

上一讲我们深入研究了 Vue 里的 compiler-dom 和 compiler-core 的流程，相信学完之后，你已经对编译原理的基础知识很熟悉了。

这时候你肯定会有一个疑问，AST、transform、generate 这些概念以前工作中也没遇见过，难道学了 this 就只能面试用吗？当然不是，编译原理作为计算机世界的一个重要的学科，除了探究原理和源码之外，我们工作中也有很多地方可以用到。



从宏观视角来看，编译原理实现的功能就是代码之间的转换。哪怕我们只是掌握了入门知识，也能可以实现 Vue 中 template 到 render 函数转化这样的功能。

现在的前端发展，很大程度上离不开编译原理在前端圈的落地实践，只要是我们想做自动化代码转化的地方，都可以看到编译的身影。

举个例子，Babel 把 ES6 中的新语法转换成低版本浏览器支持的语法，我们才能在项目中愉快地使用箭头函数等特性，把浏览器的兼容性交给 Babel 来处理，甚至现在社区内还出现了 gogocode 这种把 Vue 2 代码转换成 Vue 3 代码的工具。

在工作中我们可以借助 Babel 和 vite 提供給我們的能力，parse，transform，generate 等代码都不需要我們自己实现，只需要考虑代码转换的逻辑就可以了，下面我給你举几个小例子。


vite 插件

首先我們在項目中使用了 script setup 来组织我們的代码，虽然组件引入之后有了自动注册的功能，但是每一个组件内部都肯定要用到 ref、computed 等 Vue 提供的 API。我們還想要多一步，項目大了只引入 ref 的语句就写了几百行，就会非常地繁琐，这时候就可以使用编译的思想来解决这个问题。

首先 ref、computed、watch 等 Vue 提供的 API，我們在后面的代码调用可以通过正则匹配的方式，完全可以分析出来当前组件依赖的 API 有哪些。这样，我們就可以在组件执行之前自动导入这些 API。

我們在 weiyouyi 項目中使用 vite 插件的形式来完成这个工作。社区内已经有可用的 [🔗 auto-imput](#) 插件了，不过这里为了加深对技术的理解，咱们还是自己来实现一个。

首先我們进入到根目录下的 vite.config.js 文件中，导入 autoPlugin 插件后，配置在 vite 的 plugins 插件中。

 复制代码

```
1 import vue from '@vitejs/plugin-vue'
2 import autoPlgin from './src/auto-import'
3 export default defineConfig({
4   plugins: [vue(),autoPlgin()]
5 })
6
```

然后我们来实现 `autoPlugin` 函数，`vite` 的插件开发文档你可以在 [官网](#) 中查询，这里就不赘述了。

我们直接看代码，我们先定义了 `Vue 3` 提供的 `API` 数组，有 `ref`、`computed` 等等。然后，`autoImportPlugin` 函数对外导出一个对象，`transform` 函数就是核心要实现的逻辑。

这里的 `helper` 和我们在 32 讲中的工具函数实现逻辑一致，通过 `new RegExp` 创建每个函数匹配的正则。如果匹配到对应的 `API`，就把 `API` 的名字加入到 `helper` 集合中，最后在 `script setup` 的最上方加入一行 `import` 语句。

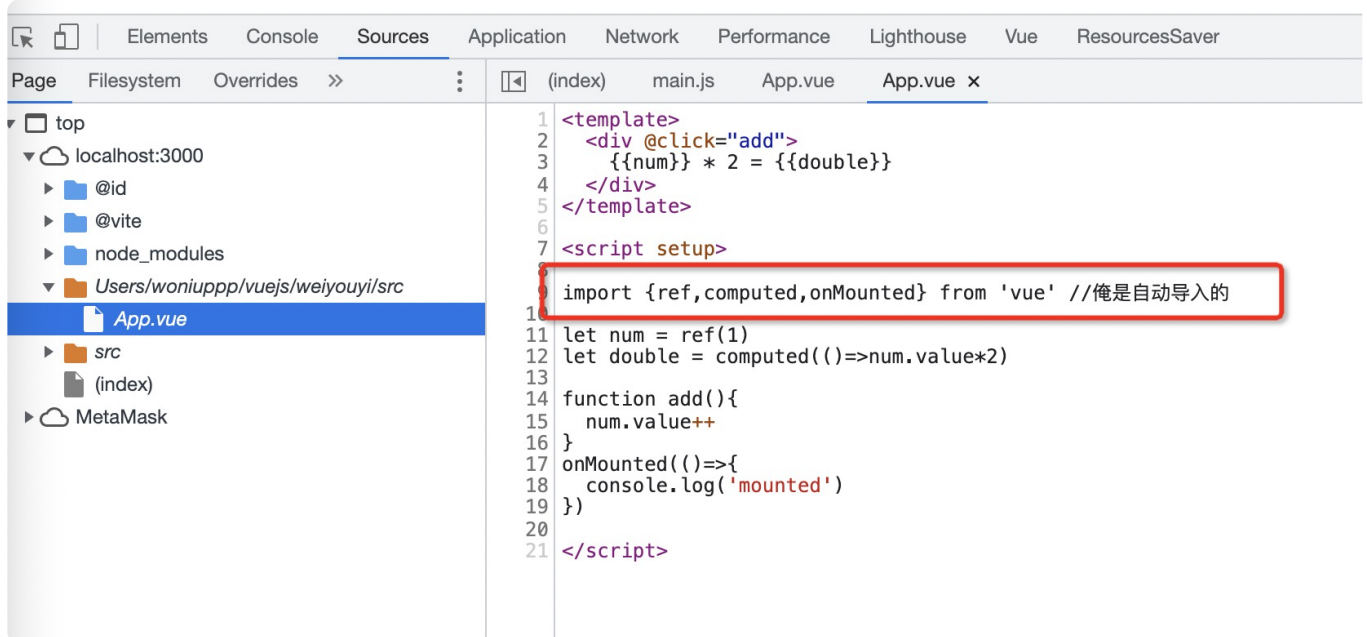
 复制代码

```
1
2  const vue3 = [
3    'ref',
4    'computed',
5    'reactive',
6    'onMounted',
7    'watchEffect',
8    'watch'
9  ] // 还有很多....
10
11 export default function autoImportPlugin() {
12   return {
13     name: 'vite-plugin-auto-import', // 必须的，将会在 warning 和 error 中显示
14     enforce: 'pre',
15     transform(code, id) {
16       vueReg = /\.vue$/
17       if (vueReg.test(id)) {
18         const helpers = new Set()
19         vue3.forEach(api => {
20           const reg = new RegExp(api + "(.*)")
21           if (reg.test(code)) {
22             helpers.add(api)
23           }
24         })
25         return code.replace('<script setup>', `
```

接着，我们在项目的 src 目录下新建 App.vue。下面的代码实现了一个简易的累加器，并且还会在 onMount 之后打印一条信息，这里的 ref、computed 和 onMounted 都是没有导入的。我们在浏览器就能看到页面可以正常显示，这时我们在浏览器调试窗口的 sources 页面中，就可以看到 App.vue 的代码已经自动加上了 import 语句。

[复制代码](#)

```
1 <template>
2   <div @click="add">
3     {{num}} * 2 = {{double}}
4   </div>
5 </template>
6
7 <script setup>
8   let num = ref(1)
9   let double = computed(()=>num.value*2)
10
11   function add(){
12     num.value++
13   }
14   onMounted(()=>{
15     console.log('mounted')
16   })
17
18 </script>
```



这里的代码都是硬编码实现的，逻辑也比较简单。不过，实际场景中判断 ref 等 API 调用的正则和导入 import 的方式，都不会这么简单。如果我们自己每次都写一个 parse 模块

比较麻烦，所以我们实际开发中会借助现有的工具对代码进行解析，而代码转换的场景下最成熟的工具就是 Babel。

Babel

我们在项目中异步的任务有很多，经常使用 `async+ await` 的语法执行异步任务，比如网络数据的获取。但 **await 是异步任务**，如果报错，我们需要使用 `try catch` 语句进行错误处理，每个 `catch` 语句都是一个打印语句会让代码变得冗余，但我们有了代码转化的思路后，这一步就能用编译的思路自动来完成。

首先我们在根目录的 `src/main.js` 中新增下面代码，我们使用 `delyError` 函数模拟异步的任务报错，在代码中使用 `await` 来模拟异步任务。

这里我们希望每个 `await` 都能跟着一个 `try` 代码，在 `catch` 中能够打印错误消息提示的同时，还能够使用调用错误监控的函数，把当前错误信息发给后端服务器进行报警，当然也可以打印一个自动去 `stackoverflow` 查询的链接。

 复制代码

```
1 function delyError(message){
2   return new Promise((resolve,reject)=>{
3     setTimeout(()=>{
4       reject({message})
5     },1000)
6   })
7 }
8 async function test(){
9   await delyError('ref is not defined')
10 }
11 // 我们期望的代码
12 async function test(){
13   try{
14     await delyError('ref is not defined')
15   }catch(e){
16     console.error(e.message)
17     _errorTrack(e.message,location.pathname)
18     console.log('https://stackoverflow.com/search?q=[js]+'+encodeURIComponent(e.messag
19   })
20
21 }
22 test()
```

页面中 `await` 语句变多了之后，手动替换的成本就比较高，我们可以继续使用 `vite` 的插件来实现。这次我们就是用 `Babel` 提供好的代码解析能力对代码进行转换。`Babel` 都提供了哪些 API，你可以在 [Babel 的官网](#) 进行深入学习。

`Babel` 提供了完整的编译代码的功能后函数，包括 AST 的解析、语义分析、代码生成等，我们可以通过下面的函数去实现自己的插件。

`@babel/parser` 提供了代码解析的能力，能够把 `js` 代码解析成 AST，代码就从字符串变成了树形结构，方便我们进行操作；

`@babel/traverse` 提供了遍历 AST 的能力，我们可以从 `traverse` 中获取每一个节点的信息后去修改它；

`@babel/types` 提供了类型判断的函数，我们可以很方便的判断每个节点的类型；

`@babel/core` 提供了代码转化的能力。

下面的代码中我们实现了 `vite-plugin-auto-try` 插件，由 `babel/parser` 解析成为 AST，通过 `traverse` 遍历整个 AST 节点，配置的 `AwaitExpression` 会识别出 AST 中的 `await` 调用语句，再用 `isTryStatement` 判断 `await` 外层是否已经包裹了 `try` 语句。如果没有 `try` 语句的话，就使用 `tryStatement` 函数生成新的 AST 节点。

这个 AST 包裹当前的节点，并且我们在内部加上了 `stackoverflow` 链接的打印。最后，使用 `babel/core` 提供的 `transformFromAstSync` 函数，把优化后的 AST 生成新的 JavaScript 代码，自动新增 `try` 代码的插件就实现了。

 复制代码

```
1
2
3 import { parse } from '@babel/parser'
4 import traverse from '@babel/traverse'
5 import {
6   isTryStatement,
7   tryStatement,
8   isBlockStatement,
9   catchClause,
10  identifier,
11  blockStatement,
12 } from '@babel/types'
13 import { transformFromAstSync } from '@babel/core'
14
```




```
15 const catchStatement = parse(`
16   console.error(err)
17   console.log('https://stackoverflow.com/search?q=[js]+'+encodeURIComponent(err.message
18 `).program.body
19
20 export default function autoImportPlugin() {
21   return {
22     name: 'vite-plugin-auto-try', // 必须的, 将会在 warning 和 error 中显示
23     enforce: 'pre',
24     transform(code, id) {
25       fileReg = /\.js$/
26       if (fileReg.test(id)) {
27         const ast = parse(code, {
28           sourceType: 'module'
29         })
30         traverse(ast, {
31           AwaitExpression(path) {
32             console.log(path)
33             if (path.findParent((path) => isTryStatement(path.node))) {
34               // 已经有try了
35               return
36             }
37             // isBlockStatement 是否函数体
38             const blockParentPath = path.findParent((path) => isBlockStatement
39             const tryCatchAst = tryStatement(
40               blockParentPath.node,
41               // ast中新增try的ast
42               catchClause(
43                 identifier('err'),
44                 blockStatement(catchStatement),
45               )
46             )
47             // 使用有try的ast替换之前的ast
48             blockParentPath.replaceWithMultiple([tryCatchAst])
49
50           }
51         })
52         // 生成代码, generate
53         code = transformFromAstSync(ast, "", {
54           configFile: false
55         }).code
56
57         return code
58       }
59       return code
60     }
61   }
62 }
```

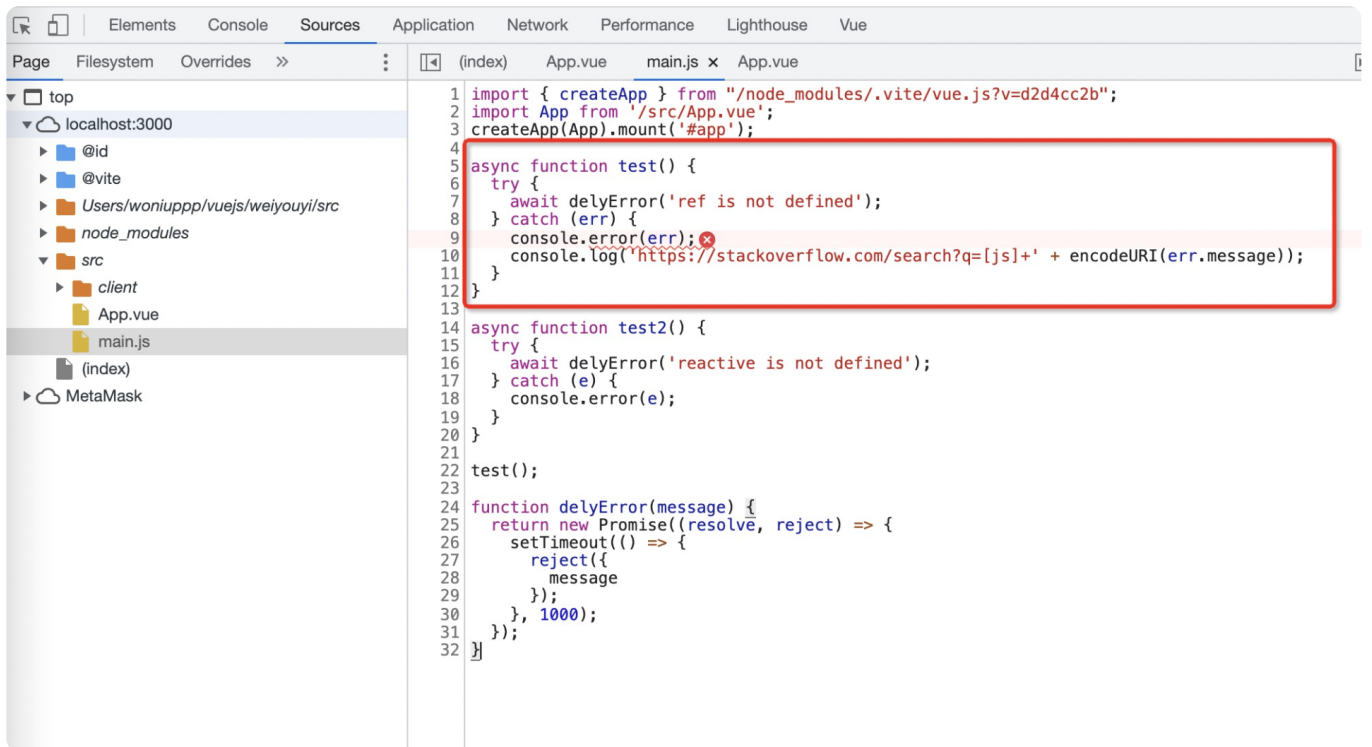
然后，我们在根目录下的 `src/main.js` 中写入下面的代码。两个 `await` 语句一个使用 `try` 包裹，一个没有使用 `try` 包裹。

接着我们启动项目后，就来到了浏览器的调试窗口中的 `source` 页面，可以看到下图中解析后的 `main.js` 代码，现在没有 `try` 的 `await` 语句已经自动加上了 `try` 语句。

你看，**这次我们基于 `babel` 来实现，就省去了我们写正则的开发成本**。Babel 提供了一整套关于 JavaScript 中语句的转化函数，有兴趣的同学可以去 Babel 官网了解。

 复制代码

```
1 import { createApp } from "vue";
2 import App from './App.vue'
3
4 createApp(App)
5   .mount('#app')
6
7 async function test(){
8   await delyError('ref is not defined')
9 }
10
11 async function test2(){
12   try{
13     await delyError('reactive is not defined')
14   }catch(e){
15     console.error(e)
16   }
17 }
18 test()
19 function delyError(message){
20   return new Promise((resolve,reject)=>{
21     setTimeout(()=>{
22       reject({message})
23     },1000)
24   })
25 }
26
```

有了 Babel 提供的能力之后，我们可以只关注于代码中需要转换的逻辑，比如我们可以使用 Babel 实现国际化，把每种语言在编译的时候自动替换语言，打包成独立的项目；也可以实现页面的自动化监控，在一些操作函数里面加入监控的代码逻辑。你可以自行发挥想象力，使用编译的思想来提高日常的开发效率。

最后我们回顾一下 Vue 中的 compiler。Vue 中的 compiler-dom 提供了 compile 函数，具体的 compile 逻辑我们在上一讲中已经详细学习了。其实我们也可以手动导入 compiler-dom 包之后，自己实现对 vue template 的解析。另外，Vue 中还提供了 @vue/compiler-sfc 包，用来实现单文件组件.vue 的解析，还有 @vue/compiler-ssr 包，它实现了服务端渲染的解析。

下一讲我们一起来手写 vite 的代码内容，我们就需要在 nodejs 中实现对 Vue 单文件组件的解析工作，实现浏览器中直接导入单文件组件的功能，敬请期待。

总结

最后我们总结一下今天学到的内容。

我们把 Vue 内部的 compiler 原理融会贯通之后，今天尝试把 template 到 render 转化过程的思想应用到实际项目中。Vue 中的 compiler 在转化的过程中还做了静态标记的优化，我们在实际开发中可以借鉴编译的思路，提高开发的效率。


我们一起回顾一下代码自动导入的操作思路。首先我们可以实现页面中 `ref`、`computed` 的 API 的自动化导入，在 `vite` 插件的 `transform` 函数中获取到待转换的代码，通过对代码的内容进行正则匹配，实现如果出现了 `ref`，`computed` 等函数的调用，我们可以把这些依赖的函数收集在 `helper` 中。最终在 `script setup` 标签之前新增 `import` 语句来导入依赖的 API，最终就可以实现代码的自动导入。

实际开发中，我们可以把使用到的组件库 `Element3`，工具函数 `vueuse` 等框架都进行语法的解析，实现函数和组件的自动化导入和按需加载。这样能在提高开发效率的同时，也提高我们书写 `vite` 插件的能力。

思考题

最后留一个思考题吧，你觉得在工作项目中有哪里需要用到代码转化的思路呢？欢迎在评论区分享你的答案，也欢迎你把这一讲的内容分享给你的同事和朋友们，我们下一讲再见！

分享给需要的人，Ta 订阅后你可得 **20 元现金奖励**

 生成海报并分享

 赞 2  提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 33 | 编译原理（中）：Vue Compiler 模块全解析

下一篇 35 | Vite 原理：写一个迷你的 Vite

更多课程推荐

陈天 · Rust 编程第一课

实战驱动，快速上手 Rust

陈天

Tubi TV 研发副总裁



涨价倒计时 🕒

今日订阅 **¥89**，1月12日涨价至 **¥199**

精选留言 (5)

💬 写留言



海阔天空

2022-01-10

厉害厉害，以前用得比较多的就是css的编译，less 函数的编译处理兼容性问题等。部分用到登录信息的处理。



费城的二鹏

2022-01-09

很多魔板代码都可以通过代码转化的方式实现，我们的网络请求代码非常固定，打算试试用这种方式减少模板代码



james

2022-01-08

不错不错



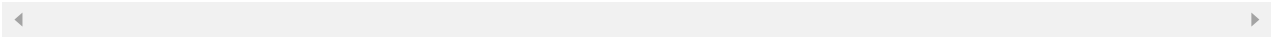
SjmBreadrain



2022-01-08

除了留言之外还有别的互动方式不？

作者回复: 课程介绍页有个微信群



Johnson

2022-01-07

很实用。 😊

