上一节课我们分析了 AST 节点转换的过程，也知道了 AST 节点转换的作用是通过语法分析，创建了语义和信息更加丰富的代码生成节点 codegenNode，便于后续生成代码。

那么这一节课，我们就来分析整个编译的过程的最后一步——代码生成的实现原理。

同样的，代码生成阶段由于要处理的场景很多，所以代码也非常多而复杂。为了方便你理解它的核心流程，我们还是通过这个示例来演示整个代码生成的过程：

```
<div>

<hello v-if="flag"></hello>

<div v-else>

<p>hello {{ msg + test }}</p>

<p>static</p>

<p>static</p>

</div>

</div>
```

代码生成的结果是和编译配置相关的，你可以打开官方提供的模板导出工具平台，点击右上角的 Options 修改编译配置。为了让你理解核心的流程，这里我只分析一种配置方案，当然当你理解整个编译核心流程后，也可以修改这些配置分析其他的分支逻辑。

我们分析的编译配置是：mode 为 module，prefixIdentifiers 开启，hoistStatic 开启，其他配置均不开启。

为了让你有个大致印象，我们先来看一下上述例子生成代码的结果：

```
import { resolveComponent as _resolveComponent, createVNode as _createVNode,
createCommentVNode as _createCommentVNode, toDisplayString as _toDisplayString,
openBlock as _openBlock, createBlock as _createBlock } from "vue"

const _hoisted_1 = { class: "app" }

const _hoisted_2 = { key: 1 }

const _hoisted_3 = _createVNode("p", null, "static", -1 )
```

```
const _hoisted_4 = _createVNode("p", null, "static", -1 )

export function render(_ctx, _cache) {

const _component_hello = _resolveComponent("hello")

return (_openBlock(), _createBlock("div", _hoisted_1, [

    (_ctx.flag)

      ? _createVNode(_component_hello, { key: 0 })

      : (_openBlock(), _createBlock("div", _hoisted_2, [

          _createVNode("p", null, "hello " + _toDisplayString(_ctx.msg +
_ctx.test), 1 ),

          _hoisted_3,

          _hoisted_4

        ]))

  ]))

}
```

示例的模板是如何转换生成这样的代码的？在 AST 转换后，会执行 generate 函数生成代码：

```
return generate(ast, extend({}, options, {

  prefixIdentifiers

}))
```

generate 函数的输入就是转换后的 AST 根节点，我们看一下它的实现：

```
function generate(ast, options = {}) {
```

```
const context = createCodegenContext(ast, options);

const { mode, push, prefixIdentifiers, indent, deindent, newline, scopeId, ssr }
= context;

const hasHelpers = ast.helpers.length > 0;

const useWithBlock = !prefixIdentifiers && mode !== 'module';

const genScopeId = scopeId != null && mode === 'module';

if ( mode === 'module') {

    genModulePreamble(ast, context, genScopeId);

  }

else {

    genFunctionPreamble(ast, context);

  }

if (!ssr) {

    push(`function render(_ctx, _cache) {`);

  }

else {

    push(`function ssrRender(_ctx, _push, _parent, _attrs) {`);

  }

  indent();

if (useWithBlock) {
```

```
    push(`with (_ctx) {`);

    indent();

if (hasHelpers) {

      push(`const { ${ast.helpers

        .map(s => `${helperNameMap[s]}: _${helperNameMap[s]}`)

        .join(', ')} } = _Vue`);

      push(`\n`);

      newline();

    }

  }

if (ast.components.length) {

    genAssets(ast.components, 'component', context);

if (ast.directives.length || ast.temps > 0) {

      newline();

    }

  }

if (ast.directives.length) {

    genAssets(ast.directives, 'directive', context);

if (ast.temps > 0) {
```

```
      newline();

    }

  }

if (ast.temps > 0) {

    push(`let `);

for (let i = 0; i < ast.temps; i++) {

      push(`${i > 0 ? `, ` : ``}_temp${i}`);

    }

  }

if (ast.components.length || ast.directives.length || ast.temps) {

    push(`\n`);

    newline();

  }

if (!ssr) {

    push(`return `);

  }

if (ast.codegenNode) {

    genNode(ast.codegenNode, context);

  }
```

```
else {

    push(`null`);

  }

if (useWithBlock) {

    deindent();

    push(`}`);

  }

  deindent();

  push(`}`);

return {

    ast,

    code: context.code,

    map: context.map ? context.map.toJSON() : undefined

  };

}
```

generate 主要做五件事情：创建代码生成上下文，生成预设代码，生成渲染函数，生成资源声明代码，以及生成创建 VNode 树的表达式。接下来，我们就依次详细分析这几个流程。

创建代码生成上下文

首先，是通过执行 createCodegenContext 创建代码生成上下文，我们来看它的实现：

```
function createCodegenContext(ast, { mode = 'function', prefixIdentifiers = mode
=== 'module', sourceMap = false, filename = `template.vue.html`, scopeId = null,
optimizeBindings = false, runtimeGlobalName = `Vue`, runtimeModuleName = `vue`,
ssr = false }) {
```

```
const context = {

    mode,

    prefixIdentifiers,

    sourceMap,

    filename,

    scopeId,

    optimizeBindings,

    runtimeGlobalName,

    runtimeModuleName,

    ssr,

    source: ast.loc.source,

    code: ``,

    column: 1,

    line: 1,

    offset: 0,

    indentLevel: 0,

    pure: false,

    map: undefined,

    helper(key) {
```

```
return `_${helperNameMap[key]}`

    },

    push(code) {

      context.code += code

    },

    indent() {

      newline(++context.indentLevel)

    },

    deindent(withoutNewLine = false) {

if (withoutNewLine) {

        --context.indentLevel

      }

else {

        newline(--context.indentLevel)

      }

    },

    newline() {

      newline(context.indentLevel)

    }

  }
```

```
function newline(n) {

    context.push('\n' + `  `.repeat(n))

  }

return context

}
```

这个上下文对象 context 维护了 generate 过程的一些配置，比如 mode、prefixIdentifiers；也维护了 generate 过程的一些状态数据，比如当前生成的代码 code，当前生成代码的缩进 indentLevel 等。

此外，context 还包含了在 generate 过程中可能会调用的一些辅助函数，接下来我会介绍几个常用的方法，它们会在整个代码生成节点过程中经常被用到。

- `push(code)`，就是在当前的代码 context.code 后追加 code 来更新它的值。
- `indent()`，它的作用就是增加代码的缩进，它会让上下文维护的代码缩进 context.indentLevel 加 1，内部会执行 newline 方法，添加一个换行符，以及两倍 indentLevel 对应的空格来表示缩进的长度。
- `deindent()`，和 indent 相反，它会减少代码的缩进，让上下文维护的代码缩进 context.indentLevel 减 1，在内部会执行 newline 方法去添加一个换行符，并减少两倍 indentLevel 对应的空格的缩进长度。

上下文创建完毕后，接下来就到了真正的代码生成阶段，在分析的过程中我会结合示例讲解，让你更直观地理解整个代码的生成过程，我们先来看生成预设代码。

生成预设代码

因为 mode 是 module，所以会执行 genModulePreamble 生成预设代码，我们来看它的实现：

```
function genModulePreamble(ast, context, genScopeId) {

const { push, newline, optimizeBindings, runtimeModuleName } = context

if (ast.helpers.length) {

if (optimizeBindings) {

      push(`import { ${ast.helpers

        .map(s => helperNameMap[s])

        .join(', ')} } from ${JSON.stringify(runtimeModuleName)}\n`)
```

```
      push(`\n
        .map(s => `_${helperNameMap[s]} = ${helperNameMap[s]}`)
        .join(', ')}\n`)
    }
else {
      push(`import { ${ast.helpers
        .map(s => `${helperNameMap[s]} as _${helperNameMap[s]}`)
        .join(', ')} } from ${JSON.stringify(runtimeModuleName)}\n`)
    }
  }
  genHoists(ast.hoists, context)
  newline()
  push(`export `)
}
```

下面我们结合前面的示例来分析这个过程，此时 genScopeId 为 false，所以相关逻辑我们可以不看。ast.helpers 是在 transform 阶段通过 context.helper 方法添加的，它的值如下：

```
Symbol(resolveComponent),
Symbol(createVNode),
Symbol(createCommentVNode),
Symbol(toDisplayString),
```

```
Symbol(openBlock),

Symbol(createBlock)

]
```

ast.helpers 存储了 Symbol 对象的数组，我们可以从 helperNameMap 中找到每个 Symbol 对象对应的字符串，helperNameMap 的定义如下：

```
const helperNameMap = {

  [FRAGMENT]: `Fragment`,

  [TELEPORT]: `Teleport`,

  [SUSPENSE]: `Suspense`,

  [KEEP_ALIVE]: `KeepAlive`,

  [BASE_TRANSITION]: `BaseTransition`,

  [OPEN_BLOCK]: `openBlock`,

  [CREATE_BLOCK]: `createBlock`,

  [CREATE_VNODE]: `createVNode`,

  [CREATE_COMMENT]: `createCommentVNode`,

  [CREATE_TEXT]: `createTextVNode`,

  [CREATE_STATIC]: `createStaticVNode`,

  [RESOLVE_COMPONENT]: `resolveComponent`,

  [RESOLVE_DYNAMIC_COMPONENT]: `resolveDynamicComponent`,

  [RESOLVE_DIRECTIVE]: `resolveDirective`,
```

```
  [WITH_DIRECTIVES]: `withDirectives`,
  [RENDER_LIST]: `renderList`,
  [RENDER_SLOT]: `renderSlot`,
  [CREATE_SLOTS]: `createSlots`,
  [TO_DISPLAY_STRING]: `toDisplayString`,
  [MERGE_PROPS]: `mergeProps`,
  [TO_HANDLERS]: `toHandlers`,
  [CAMELIZE]: `camelize`,
  [SET_BLOCK_TRACKING]: `setBlockTracking`,
  [PUSH_SCOPE_ID]: `pushScopeId`,
  [POP_SCOPE_ID]: `popScopeId`,
  [WITH_SCOPE_ID]: `withScopeId`,
  [WITH_CTX]: `withCtx`
}
```

由于 optimizeBindings 是 false，所以会执行如下代码：

```
push(`import { ${ast.helpers
  .map(s => `${helperNameMap[s]} as _${helperNameMap[s]}`)
  .join(', ')} } from ${JSON.stringify(runtimeModuleName)}\n`)
}
```

最终会生成这些代码，并更新到 context.code 中：

```
import { resolveComponent as _resolveComponent, createVNode as _createVNode,
createCommentVNode as _createCommentVNode, toDisplayString as _toDisplayString,
openBlock as _openBlock, createBlock as _createBlock } from "vue"
```

通过生成的代码，我们可以直观地感受到，这里就是从 Vue 中引入了一些辅助方法，那么为什么需要引入这些辅助方法呢，这就和 Vue.js 3.0 的设计有关了。

在 Vue.js 2.x 中，创建 VNode 的方法比如 $createElement、_c 这些都是挂载在组件的实例上，在生成渲染函数的时候，直接从组件实例 vm 中访问这些方法即可。

而到了 Vue.js 3.0，创建 VNode 的方法 createVNode 是直接通过模块的方式导出，其它方法比如 resolveComponent、openBlock，都是类似的，所以我们首先需要生成这些 import 声明的预设代码。

我们接着往下看，ssrHelpers 是 undefined，imports 的数组长度为空，genScopeId 为 false，所以这些内部逻辑都不会执行，接着执行 genHoists 生成静态提升的相关代码，我们来看它的实现：

```
function genHoists(hoists, context) {

if (!hoists.length) {

return

  }

  context.pure = true

const { push, newline } = context

newline()

  hoists.forEach((exp, i) => {

if (exp) {

      push(`const _hoisted_${i + 1} = `)

      genNode(exp, context)

      newline()

    }
```

```
  })

  context.pure = false

}
```

首先通过执行 newline 生成一个空行，然后遍历 hoists 数组，生成静态提升变量定义的方法。此时 hoists 的值是这样的：

```
  {

"type": 15,

"properties": [

      {

"type": 16,

"key": {

"type": 4,

"isConstant": false,

"content": "class",

"isStatic": true

        },

"value": {

"type": 4,

"isConstant": false,

"content": "app",
```

```
"isStatic": true

        }

      }

    ]

  },

  {

"type": 15,

"properties": [

      {

"type": 16,

"key": {

"type": 4,

"isConstant": false,

"content": "key",

"isStatic": true

        },

"value": {

"type": 4,

"isConstant": false,
```

```
"content": "1",

"isStatic": false

        }

      }

    ]

  },

  {

"type": 13,

"tag": "\"p\"",

"children": {

"type": 2,

"content": "static"

    },

"patchFlag": "-1 /* HOISTED */",

"isBlock": false,

"disableTracking": false

  },

  {

"type": 13,
```

```
"tag": "\"p\"",

"children": {

"type": 2,

"content": "static",

    },

"patchFlag": "-1 /* HOISTED */",

"isBlock": false,

"disableTracking": false,

  }

]
```

这里，hoists 数组的长度为 4，前两个都是 JavaScript 对象表达式节点，后两个是 VNodeCall 节点，通过 genNode 我们可以把这些节点生成对应的代码，这个方法我们后续会详细说明，这里先略过。

然后通过遍历 hoists 我们生成如下代码：

```
import { resolveComponent as _resolveComponent, createVNode as _createVNode,
createCommentVNode as _createCommentVNode, toDisplayString as _toDisplayString,
openBlock as _openBlock, createBlock as _createBlock } from "vue"


const _hoisted_1 = { class: "app" }


const _hoisted_2 = { key: 1 }


const _hoisted_3 = _createVNode("p", null, "static", -1 )


const _hoisted_4 = _createVNode("p", null, "static", -1 )
```

可以看到，除了从 Vue 中导入辅助方法，我们还创建了静态提升的变量。

我们回到 genModulePreamble，接着会执行`newline()`和`push(export )`，非常好理解，也就是添加了一个空行和 export 字符串。

至此，预设代码生成完毕，我们就得到了这些代码：

```
import { resolveComponent as _resolveComponent, createVNode as _createVNode,
createCommentVNode as _createCommentVNode, toDisplayString as _toDisplayString,
openBlock as _openBlock, createBlock as _createBlock } from "vue"


const _hoisted_1 = { class: "app" }


const _hoisted_2 = { key: 1 }


const _hoisted_3 = _createVNode("p", null, "static", -1 )


const _hoisted_4 = _createVNode("p", null, "static", -1 )


export
```

生成渲染函数

接下来，就是生成渲染函数了，我们回到 generate 函数：

```
if (!ssr) {


push(`function render(_ctx, _cache) {`);


}


else {


push(`function ssrRender(_ctx, _push, _parent, _attrs) {`);


}


indent();
```

由于 ssr 为 false, 所以生成如下代码：

```
import { resolveComponent as _resolveComponent, createVNode as _createVNode,
createCommentVNode as _createCommentVNode, toDisplayString as _toDisplayString,
openBlock as _openBlock, createBlock as _createBlock } from "vue"


const _hoisted_1 = { class: "app" }
```

```
const _hoisted_2 = { key: 1 }

const _hoisted_3 = _createVNode("p", null, "static", -1 )

const _hoisted_4 = _createVNode("p", null, "static", -1 )

export function render(_ctx, _cache) {
```

注意，**这里代码的最后一行有 2 个空格的缩进。**

另外，由于 useWithBlock 为 false，所以我们也不需生成 with 相关的代码。而且，这里我们创建了 render 的函数声明，接下来的代码都是在生成 render 的函数体。

生成资源声明代码

在 render 函数体的内部，我们首先要生成资源声明代码：

```
if (ast.components.length) {

  genAssets(ast.components, 'component', context);

if (ast.directives.length || ast.temps > 0) {

    newline();

  }

}

if (ast.directives.length) {

  genAssets(ast.directives, 'directive', context);

if (ast.temps > 0) {

    newline();

  }

}
```

```
if (ast.temps > 0) {

  push(`let `);

for (let i = 0; i < ast.temps; i++) {

    push(`${i > 0 ? `, ` : ``}_temp${i}`);

  }

}
```

在我们的示例中，directives 数组长度为 0，temps 的值是 0，所以自定义指令和临时变量代码生成的相关逻辑跳过，而这里 components 的值是 `["hello"]`。

接着就通过 genAssets 去生成自定义组件声明代码，我们来看一下它的实现：

```
function genAssets(assets, type, { helper, push, newline }) {

const resolver = helper(type === 'component' ? RESOLVE_COMPONENT :
RESOLVE_DIRECTIVE)

for (let i = 0; i < assets.length; i++) {

const id = assets[i]

    push(`const ${toValidAssetId(id, type)} = ${resolver}
(${JSON.stringify(id)})`)

if (i < assets.length - 1) {

      newline()

    }

  }

}
```

这里的 helper 函数就是从前面提到的 helperNameMap 中查找对应的字符串，对于 component，返回的就是 resolveComponent。

接着会遍历 assets 数组，生成自定义组件声明代码，在这个过程中，它们会把变量通过 toValidAssetId 进行一层包装：

```
function toValidAssetId(name, type) {


return `_${type}_${name.replace(/[^\w]/g, '_')}`;


}
```

比如 hello 组件，执行 toValidAssetId 就变成了 _component_hello。

因此对于我们的示例而言，genAssets 后生成的代码是这样的：

```
import { resolveComponent as _resolveComponent, createVNode as _createVNode,
createCommentVNode as _createCommentVNode, toDisplayString as _toDisplayString,
openBlock as _openBlock, createBlock as _createBlock } from "vue"


const _hoisted_1 = { class: "app" }


const _hoisted_2 = { key: 1 }


const _hoisted_3 = _createVNode("p", null, "static", -1 )


const _hoisted_4 = _createVNode("p", null, "static", -1 )


export function render(_ctx, _cache) {


const _component_hello = _resolveComponent("hello")
```

这很好理解，通过 resolveComponent，我们就可以解析到注册的自定义组件对象，然后在后面创建组件 vnode 的时候当做参数传入。

回到 generate 函数，接下来会执行如下代码：

```
if (ast.components.length || ast.directives.length || ast.temps) {


  push(`\n`);


  newline();
```

```
}

if (!ssr) {

  push(`return `);

}
```

这里是指，如果生成了资源声明代码，则在尾部添加一个换行符，然后再生成一个空行，并且如果不是 ssr，则再添加一个 return 字符串，此时得到的代码结果如下：

```
import { resolveComponent as _resolveComponent, createVNode as _createVNode,
createCommentVNode as _createCommentVNode, toDisplayString as _toDisplayString,
openBlock as _openBlock, createBlock as _createBlock } from "vue"

const _hoisted_1 = { class: "app" }

const _hoisted_2 = { key: 1 }

const _hoisted_3 = _createVNode("p", null, "static", -1 )

const _hoisted_4 = _createVNode("p", null, "static", -1 )

export function render(_ctx, _cache) {

const _component_hello = _resolveComponent("hello")

return
```

好的，我们就先分析到这里，下节课继续来看生成创建 VNode 树的表达式的过程。

> 本节课的相关代码在源代码中的位置如下：
> packages/compiler-core/src/codegen.ts