上一节课，我们已经知道了 transform 的核心流程主要有四步：创建 transform 上下文、遍历 AST 节点、静态提升以及创建根代码生成节点。这节课我们接着分析遍历 AST 节点中的 Text 节点的转换函数。

遍历 AST 节点

Text 节点转换函数

接下来，我们来看一下 Text 节点转换函数的实现：

```
const transformText = (node, context) => {

if (node.type === 0  ||

    node.type === 1  ||

    node.type === 11  ||

    node.type === 10 ) {

return () => {

const children = node.children

      let currentContainer = undefined

      let hasText = false

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

const child = children[i]

if (isText(child)) {

          hasText = true

for (let j = i + 1; j < children.length; j++) {

const next = children[j]
```

```
if (isText(next)) {

if (!currentContainer) {

                currentContainer = children[i] = {

                  type: 8 ,

                  loc: child.loc,

                  children: [child]

                }

              }

              currentContainer.children.push(` + `, next)

              children.splice(j, 1)

              j--

            }

else {

              currentContainer = undefined

break

            }

          }

        }

      }
```

```
if (!hasText ||

        (children.length === 1 &&

          (node.type === 0  ||

            (node.type === 1  &&

              node.tagType === 0 )))) {

return

      }

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

const child = children[i]

if (isText(child) || child.type === 8 ) {

const callArgs = []

if (child.type !== 2  || child.content !== ' ') {

            callArgs.push(child)

          }

if (!context.ssr && child.type !== 2 ) {

            callArgs.push(`${1 } ]} */`)

          }

          children[i] = {

            type: 12 ,
```

```
            content: child,

            loc: child.loc,

            codegenNode: createCallExpression(context.helper(CREATE_TEXT),
callArgs)

          }

        }

      }

    }

  }

}
```

transformText 函数只处理根节点、元素节点、 v-for 以及 v-if 分支相关的节点，它也会返回一个退出函数，因为 transformText 要保证所有表达式节点都已经被处理才执行转换逻辑。

transformText 主要的目的就是合并一些相邻的文本节点，然后为内部每一个文本节点创建一个代码生成节点。

在内部，静态文本节点和动态插值节点都被看作是一个文本节点，所以函数首先遍历节点的子节点，然后把子节点中的相邻文本节点合并成一个。

比如示例中的文本节点：`<p>hello {{ msg + test }}</p>`。

在转换之前，p 节点对应的 children 数组有两个元素，第一个是纯文本节点，第二个是一个插值节点，这个数组也是前面提到的表达式节点转换后的结果：

```
  {

"type": 2,

"content": "hello ",

  },

  {

"type": 5,
```

```
"content": {
"type": 8,
"children": [
        {
"type": 4,
"isConstant": false,
"content": "_ctx.msg",
"isStatic": false
        },
" + ",
        {
"type": 4,
"isConstant": false,
"content": "_ctx.test",
"isStatic": false
        }
      ],
"identifiers": []
    }
```

```
  }

]
```

转换后，这两个文本节点被合并成一个复合表达式节点，结果如下：

```
  {

"type": 8,

"children": [

      {

"type": 2,

"content": "hello ",

      },

" + ",

      {

"type": 5,

"content": {

"type": 8,

"children": [

            {

"type": 4,

"isConstant": false,
```

```
"content": "_ctx.msg",

"isStatic": false

            },

" + ",

            {

"type": 4,

"isConstant": false,

"content": "_ctx.test",

"isStatic": false

            }

          ],

"identifiers": []

        }

      }

    ]

  }

]
```

合并完子文本节点后，接着判断如果是一个只带有单个文本子元素的纯元素节点，则什么都不需要转换，因为这种情况在运行时可以直接设置元素的 textContent 来更新文本。

最后就是去处理节点包含文本子节点且多个子节点的情况，举个例子：

```
<p>
```

```
  hello {{ msg + test }}

  <a href="foo"/>

  hi

</p>
```

上述 p 标签的子节点经过前面的文本合并流程后，还有 3 个子节点。针对这种情况，我们可以遍历子节点，找到所有的文本节点或者是复合表达式节点，然后为这些子节点通过 createCallExpression 创建一个调用函数表达式的代码生成节点。

我们来看 createCallExpression 的实现：

```
function createCallExpression(callee, args = [], loc = locStub) {

return {

    type: 14 ,

    loc,

    callee,

    arguments: args

  }

}
```

createCallExpression 的实现很简单，就是返回一个类型为 JS_CALL_EXPRESSION 的对象，它包含了执行的函数名和参数。

这里，针对我们创建的函数表达式所生成的节点，它对应的函数名是 createTextVNode，参数 callArgs 是子节点本身 child，如果是动态插值节点，那么参数还会多一个 TEXT 的 patchFlag。

v-if 节点转换函数

接下来，我们来看一下 v-if 节点转换函数的实现：

```
const transformIf = createStructuralDirectiveTransform(/^(if|else|else-if)$/,
(node, dir, context) => {
```

```
return processIf(node, dir, context, (ifNode, branch, isRoot) => {

return () => {

    }

  })

})
```

在分析函数的实现前，我们先来看一下 v-if 节点转换的目的，为了方便你的理解，我还是通过示例来说明：

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

<div v-else>

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

<p>static</p>

<p>static</p>

</div>
```

在 parse 阶段，这个模板解析生成的 AST 节点如下：

```
  {

"children": [],

"codegenNode": undefined,

"isSelfClosing": false,

"ns": 0,

"props": [{
```

```
"type": 7,

"name": "if",

"exp": {

"type": 4,

"content": "flag",

"isConstant": false,

"isStatic": false

      },

"arg": undefined,

"modifiers": []

    }],

"tag": "hello",

"tagType": 1,

"type": 1

  },

  {

"children": [

    ],

"codegenNode": undefined,
```

```
"isSelfClosing": false,

"ns": 0,

"props": [{

"type": 7,

"name": "else",

"exp": undefined,

"arg": undefined,

"modifiers": []

    }],

"tag": "div",

"tagType": 0,

"type": 1

  }

]
```

v-if 指令用于条件性地渲染一块内容，显然上述 AST 节点对于最终去生成条件的代码而言，是不够语义化的，于是我们需要对它们做一层转换，使其成为语义化强的代码生成节点。

现在我们回过头看 transformIf 的实现，它是通过 createStructuralDirectiveTransform 函数创建的一个结构化指令的转换函数，在 Vue.js 中，v-if、v-else-if、v-else 和 v-for 这些都属于结构化指令，因为它们能影响代码的组织结构。

我们来看一下 createStructuralDirectiveTransform 的实现：

```
function createStructuralDirectiveTransform(name, fn) {

const matches = isString(name)
```

```
    ? (n) => n === name

    : (n) => name.test(n)

return (node, context) => {

if (node.type === 1 ) {

const { props } = node

if (node.tagType === 3  && props.some(isVSlot)) {

return

      }

const exitFns = []

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

const prop = props[i]

if (prop.type === 7  && matches(prop.name)) {

          props.splice(i, 1)

          i--

const onExit = fn(node, prop, context)

if (onExit)

            exitFns.push(onExit)

        }

      }

return exitFns
```

```
    }

  }

}
```

可以看到，createStructuralDirectiveTransform 接受 2 个参数，第一个 name 是指令的名称，第二个 fn 是构造转换退出函数的方法。

createStructuralDirectiveTransform 最后会返回一个函数，在我们的场景下，这个函数就是 transformIf 转换函数。

我们进一步看这个函数的实现，它只处理元素节点，这个很好理解，因为只有元素节点才会有 v-if 指令，接着会解析这个节点的 props 属性，如果发现 props 包含 if 属性，也就是节点拥有 v-if 指令，那么先从 props 删除这个结构化指令防止无限递归，然后执行 fn 获取对应的退出函数，最后将这个退出函数返回。

接着我们来看 fn 的实现，在我们这个场景下 fn 对应的是前面传入的匿名函数：

```
(node, dir, context) => {

return processIf(node, dir, context, (ifNode, branch, isRoot) => {

return () => {

    }

  })

}
```

可以看出，这个匿名函数内部执行了 processIf 函数，它会先对 v-if 和它的相邻节点做转换，然后返回一个退出函数，在它们的子节点都转换完毕后执行。

我们来看 processIf 函数的实现：

```
function processIf(node, dir, context, processCodegen) {

if (dir.name === 'if') {

const branch = createIfBranch(node, dir)

const ifNode = {
```

```
      type: 9 ,
      loc: node.loc,
      branches: [branch]
    }
    context.replaceNode(ifNode)
if (processCodegen) {
return processCodegen(ifNode, branch, true)
    }
  }
else {
  }
}
```

processIf 主要就是用来处理 v-if 节点以及 v-if 的相邻节点，比如 v-else-if 和 v-else，并且它们会走不同的处理逻辑。

我们先来看 v-if 的处理逻辑。首先，它会执行 createIfBranch 去创建一个分支节点：

```
function createIfBranch(node, dir) {
return {
    type: 10 ,
    loc: node.loc,
    condition: dir.name === 'else' ? undefined : dir.exp,
```

```
    children: node.tagType === 3  ? node.children : [node]

  }

}
```

这个分支节点很好理解，因为 v-if 节点内部的子节点可以属于一个分支，v-else-if 和 v-else 节点内部的子节点也都可以属于一个分支，而最终页面渲染执行哪个分支，这取决于哪个分支节点的 condition 为 true。

所以分支节点返回的对象，就包含了 condition 条件，以及它的子节点 children。注意，**如果节点 node 不是 template，那么 children 指向的就是这个单个 node 构造的数组。**

接下来它会创建 IF 节点替换当前节点，IF 节点拥有 branches 属性，包含我们前面创建的分支节点，显然，相对于原节点，IF 节点的语义化更强，更利于后续生成条件表达式代码。

最后它会执行 processCodegen 创建退出函数。我们先不着急去分析退出函数的创建过程，先把 v-if 相邻节点的处理逻辑分析完：

```
function processIf(node, dir, context, processCodegen) {

if (dir.name === 'if') {

  }

else {

const siblings = context.parent.children

    let i = siblings.indexOf(node)

while (i-- >= -1) {

const sibling = siblings[i]

if (sibling && sibling.type === 9 ) {

        context.removeNode()

const branch = createIfBranch(node, dir)

        sibling.branches.push(branch)
```

```
const onExit = processCodegen && processCodegen(sibling, branch, false)

        traverseNode(branch, context)

if (onExit)

          onExit()

        context.currentNode = null

      }

else {

        context.onError(createCompilerError(28 , node.loc))

      }

break

    }

  }

}
```

这段处理逻辑就是从当前节点往前面的兄弟节点遍历，找到 v-if 节点后，把当前节点删除，然后根据当前节点创建一个分支节点，把这个分支节点添加到前面创建的 IF 节点的 branches 中。此外，由于这个节点已经删除，那么需要在这里把这个节点的子节点通过 traverseNode 遍历一遍。

这么处理下来，就相当于完善了 IF 节点的信息了，IF 节点的 branches 就包含了所有分支节点了。

那么至此，进入 v-if、v-else-if、v-else 这些节点的转换逻辑我们就分析完毕了，即最终创建了一个 IF 节点，它包含了所有的分支节点。

接下来，我们再来分析这个退出函数的逻辑：

```
(node, dir, context) => {

return processIf(node, dir, context, (ifNode, branch, isRoot) => {
```

```
return () => {

if (isRoot) {

        ifNode.codegenNode = createCodegenNodeForBranch(branch, 0, context)

      }

else {

        let parentCondition = ifNode.codegenNode

while (parentCondition.alternate.type ===

19 ) {

          parentCondition = parentCondition.alternate

        }

        parentCondition.alternate = createCodegenNodeForBranch(branch,
ifNode.branches.length - 1, context)

      }

    }

  })

}
```

可以看到，当 v-if 节点执行退出函数时，会通过 createCodegenNodeForBranch 创建 IF 分支节点的 codegenNode，我们来看一下它的实现：

```
function createCodegenNodeForBranch(branch, index, context) {

if (branch.condition) {

return createConditionalExpression(branch.condition,
createChildrenCodegenNode(branch, index, context),
```

```
      createCallExpression(context.helper(CREATE_COMMENT), [

        (process.env.NODE_ENV !== 'production') ? '"v-if"' : '""',

'true'

      ]))

  }

else {

return createChildrenCodegenNode(branch, index, context)

  }

}
```

当分支节点存在 condition 的时候，比如 v-if、和 v-else-if，它通过 createConditionalExpression 返回一个条件表达式节点：

```
function createConditionalExpression(test, consequent, alternate, newline =
true) {

return {

    type: 19 ,

    test,

    consequent,

    alternate,

    newline,

    loc: locStub
```

```
  }

}
```

其中 consequent 在这里是 IF 主 branch 的子节点对应的代码生成节点，alternate 是后补 branch 子节点对应的代码生成节点。

接着，我们来看一下 createChildrenCodegenNode 的实现：

```
function createChildrenCodegenNode(branch, index, context) {

const { helper } = context

const keyProperty = createObjectProperty(`key`, createSimpleExpression(index +
'', false))

const { children } = branch

const firstChild = children[0]

const needFragmentWrapper = children.length !== 1 || firstChild.type !== 1

if (needFragmentWrapper) {

if (children.length === 1 && firstChild.type === 11 ) {

const vnodeCall = firstChild.codegenNode

injectProp(vnodeCall, keyProperty, context)

      return vnodeCall

    }

else {

return createVNodeCall(context, helper(FRAGMENT),
createObjectExpression([keyProperty]), children, `${64 } ]} */`, undefined,
undefined, true, false, branch.loc)

    }
```

```
  }

else {

const vnodeCall = firstChild

      .codegenNode;

if (vnodeCall.type === 13  &&

      (firstChild.tagType !== 1  ||

        vnodeCall.tag === TELEPORT)) {

      vnodeCall.isBlock = true

      helper(OPEN_BLOCK)

      helper(CREATE_BLOCK)

    }

    injectProp(vnodeCall, keyProperty, context)

return vnodeCall

  }

}
```

createChildrenCodegenNode 主要就是判断每个分支子节点是不是一个 vnodeCall，如果这个子节点不是组件节点的话，则把它转变成一个 BlockCall，也就是让 v-if 的每一个分支都可以创建一个 Block。

这个行为是很好理解的，因为 v-if 是条件渲染的，我们知道在某些条件下某些分支是不会渲染的，那么它内部的动态节点就不能添加到外部的 Block 中的，所以它就需要单独创建一个 Block 来维护分支内部的动态节点，这样也就构成了 Block tree。

为了直观让你感受 v-if 节点最终转换的结果，我们来看前面示例转换后的结果，最终转换生成的 IF 节点对象大致如下：

```
{

"type": 9,

"branches": [{

"type": 10,

"children": [{

"type": 1,

"tagType": 1,

"tag": "hello"

     }],

"condition": {

"type": 4,

"content": "_ctx.flag"

     }

  },{

"type": 10,

"children": [{

"type": 1,

"tagType": 0,

"tag": "hello"

     }],
```

```
"condition": {

"type": 4,

"content": "_ctx.flag"

     }

  }],

"codegenNode": {

"type": 19,

"consequent": {

"type": 13,

"tag": "_component_hello",

"children": undefined,

"directives": undefined,

"dynamicProps": undefined,

"isBlock": false,

"patchFlag": undefined

    },

"alternate": {

"type": 13,

"tag": "_component_hello",
```

```
"children": [

      ],

"directives": undefined,

"dynamicProps": undefined,

"isBlock": false,

"patchFlag": undefined

    }

  }

}
```

可以看到，相比原节点，转换后的 IF 节点无论是在语义化还是在信息上，都更加丰富，我们可以依据它在代码生成阶段生成所需的代码。

静态提升

节点转换完毕后，接下来会判断编译配置中是否配置了 hoistStatic，如果是就会执行 hoistStatic 做静态提升：

```
if (options.hoistStatic) {

  hoistStatic(root, context)

}
```

静态提升也是 Vue.js 3.0 在编译阶段设计了一个优化策略，为了便于你理解，我先举个简单的例子：

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

<p>static</p>

<p>static</p>
```

我们为它配置了 hoistStatic，经过编译后，它的代码就变成了这样：

```
import { toDisplayString as _toDisplayString, createVNode as _createVNode,
Fragment as _Fragment, openBlock as _openBlock, createBlock as _createBlock }
from "vue"


const _hoisted_1 = _createVNode("p", null, "static", -1 )


const _hoisted_2 = _createVNode("p", null, "static", -1 )


export function render(_ctx, _cache) {


return (_openBlock(), _createBlock(_Fragment, null, [


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


    _hoisted_1,


    _hoisted_2


  ], 64 ))


}
```

这里，我们先忽略 openBlock、Fragment ，我会在代码生成章节详细说明，重点看一下 _hoisted_1 和 _hoisted_2 这两个变量，它们分别对应模板中两个静态 p 标签生成的 vnode，可以发现它的创建是在 render 函数外部执行的。
这样做的好处是，不用每次在 render 阶段都执行一次 createVNode 创建 vnode 对象，直接用之前在内存中创建好的 vnode 即可。

那么为什么叫静态提升呢?

因为这些静态节点不依赖动态数据，一旦创建了就不会改变，所以只有静态节点才能被提升到外部创建。

了解以上背景知识后，我们接下来看一下静态提升的实现：

```
function hoistStatic(root, context) {


  walk(root, context, new Map(),


    isSingleElementRoot(root, root.children[0]));
```

```
}

function walk(node, context, resultCache, doNotHoistNode = false) {

  let hasHoistedNode = false

  let hasRuntimeConstant = false

const { children } = node

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

const child = children[i]

if (child.type === 1  &&

      child.tagType === 0 ) {

let staticType

if (!doNotHoistNode &&

        // 获取静态节点的类型，如果是元素，则递归检查它的子节点

        (staticType = getStaticType(child, resultCache)) > 0) {

if (staticType === 2 ) {

          hasRuntimeConstant = true

        }

        child.codegenNode.patchFlag =

          -1  + ((process.env.NODE_ENV !== 'production') ? ` ` : ``)

        child.codegenNode = context.hoist(child.codegenNode)

        hasHoistedNode = true
```

```
continue

      }

else {

const codegenNode = child.codegenNode

if (codegenNode.type === 13 ) {

const flag = getPatchFlag(codegenNode)

if ((!flag ||

             flag === 512  ||

             flag === 1 ) &&

             !hasDynamicKeyOrRef(child) &&

             !hasCachedProps()) {

const props = getNodeProps(child)

if (props) {

               codegenNode.props = context.hoist(props)

             }

           }

         }

      }

    }
```

```
else if (child.type === 12 ) {

const staticType = getStaticType(child.content, resultCache)

if (staticType > 0) {

if (staticType === 2 ) {

          hasRuntimeConstant = true

        }

        child.codegenNode = context.hoist(child.codegenNode)

        hasHoistedNode = true

      }

    }

if (child.type === 1 ) {

      walk(child, context, resultCache)

    }

else if (child.type === 11 ) {

      walk(child, context, resultCache, child.children.length === 1)

    }

else if (child.type === 9 ) {

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

        walk(child.branches[i], context, resultCache,
child.branches[i].children.length === 1)
```

```
      }

    }

  }

if (!hasRuntimeConstant && hasHoistedNode && context.transformHoist) {

    context.transformHoist(children, context, node)

  }

}
```

可以看到，hoistStatic 主要就是从根节点开始，通过递归的方式去遍历节点，只有普通元素和文本节点才能被静态提升，所以针对这些节点，这里通过 getStaticType 去获取静态类型，如果节点是一个元素类型，getStaticType 内部还会递归判断它的子节点的静态类型。

虽然有的节点包含一些动态子节点，但它本身的静态属性还是可以被静态提升的。

注意，**如果 getStaticType 返回的 staticType 的值是 2，则表明它是一个运行时常量，由于它的值在运行时才能被确定，所以是不能静态提升的。**

如果节点满足可以被静态提升的条件，节点对应的 codegenNode 会通过执行 context.hoist 修改为一个简单表达式节点：

```
function hoist(exp) {

  context.hoists.push(exp);

const identifier = createSimpleExpression(`_hoisted_${context.hoists.length}`,
false, exp.loc, true)

  identifier.hoisted = exp

return identifier

}

child.codegenNode = context.hoist(child.codegenNode)
```

改动后的 codegenNode 会在生成代码阶段帮助我们生成静态提升的相关代码。

createRootCodegen

完成静态提升后，我们来到了 AST 转换的最后一步，即**创建根节点的代码生成节点**。我们先来看一下 createRootCodegen 的实现：

```
function createRootCodegen(root, context) {

const { helper } = context;

const { children } = root;

const child = children[0];

if (children.length === 1) {

if (isSingleElementRoot(root, child) && child.codegenNode) {

const codegenNode = child.codegenNode;

if (codegenNode.type === 13 ) {

        codegenNode.isBlock = true;

        helper(OPEN_BLOCK);

        helper(CREATE_BLOCK);

      }

      root.codegenNode = codegenNode;

    }

else {

      root.codegenNode = child;

    }

  }
```

```
else if (children.length > 1) {

    root.codegenNode = createVNodeCall(context, helper(FRAGMENT), undefined,
root.children, `${64 } ]} */`, undefined, undefined, true);

  }

}
```

createRootCodegen 做的事情很简单，就是为 root 这个虚拟的 AST 根节点创建一个代码生成节点，如果 root 的子节点 children 是单个元素节点，则将其转换成一个 Block，把这个 child 的 codegenNode 赋值给 root 的 codegenNode。

如果 root 的子节点 children 是多个节点，则返回一个 fragement 的代码生成节点，并赋值给 root 的 codegenNode。

这里，创建 codegenNode 就是为了后续生成代码时使用。

createRootCodegen 完成之后，接着把 transform 上下文在转换 AST 节点过程中创建的一些变量赋值给 root 节点对应的属性，在这里可以看一下这些属性：

```
root.helpers = [...context.helpers]

root.components = [...context.components]

root.directives = [...context.directives]

root.imports = [...context.imports]

root.hoists = context.hoists

root.temps = context.temps

root.cached = context.cached
```

这样后续在代码生成节点时，就可以通过 root 这个根节点访问到这些变量了。

总结

好的，到这里我们这一节的学习就结束啦，通过这节课的学习，你应该对 AST 节点内部做了哪些转换有所了解。

如果说 parse 阶段是一个词法分析过程，构造基础的 AST 节点对象，那么 transform 节点就是语法分析阶段，把 AST 节点做一层转换，构造出语义化更强，信息更加丰富的 codegenCode，它在后续的代码生成阶段起着非常重要的作用。

最后，给你留一道思考题目，我们已经知道静态提升的好处是，针对静态节点不用每次在 render 阶段都执行一次 createVNode 创建 vnode 对象，但它有没有成本呢？为什么？欢迎你在留言区与我分享。

> 本节课的相关代码在源代码中的位置如下：
> packages/compiler-core/src/ast.ts
> packages/compiler-core/src/transform.ts
> packages/compiler-core/src/transforms/transformText.ts
> packages/compiler-core/src/transforms/vIf.ts
> packages/compiler-core/src/transforms/hoistStatic.ts