Sunday • 更新于 2022-10-19

上一节 10：基于编辑器... 12：基于编辑器... 下一节

11：基于编辑器的指令(v-xx)处理：JavaScript AST，构建 vif 转化模块（困难）

vue 内部具备非常多的指令，所以我们需要有一个统一的方法来对这些指令进行处理，在 packages/compiler-core/src/transform.ts 模块下，创建 createStructuralDirectiveTransform 方法，该方法返回一个闭包函数：

代码块

```
/**
 * 针对于指令的处理
 * @param name 正则。匹配具体的指令
 * @param fn 指令的具体处理方法，通常为闭包函数
 * @returns 返回一个闭包函数
 */
export function createStructuralDirectiveTransform(name: string | RegExp, fn) {
    const matches = isString(name)
        ? (n: string) => n === name
        : (n: string) => name.test(n)

    return (node, context) => {
        if (node.type === NodeTypes.ELEMENT) {
            const { props } = node
            // 结构的转换与 v-slot 无关
            if (node.tagType === ElementTypes.TEMPLATE && props.some(isVSlot)) {
                return
            }

            // 存储转化函数的数组
            const exitFns: any = []
            // 遍历所有的 props
            for (let i = 0; i < props.length; i++) {
                const prop = props[i]
                // 仅处理指令，并且该指令要匹配指定的正则
                if (prop.type === NodeTypes.DIRECTIVE && matches(prop.name)) {
                    // 删除结构指令以避免无限递归
                    props.splice(i, 1)
                    i--
                    // fn 会返回具体的指令函数
                    const onExit = fn(node, prop, context)
                    // 存储到数组中
                    if (onExit) exitFns.push(onExit)
                }
            }
            // 返回包含所有函数的数组
            return exitFns
        }
    }
}
```

这里使用到了一个 isVSlot 函数，我们需要在 packages/compiler-core/src/utils.ts 中创建该函数：

代码块

```
/**
 * 是否为 v-slot
```

```
export function isVSlot(p) {
    return p.type === NodeTypes.DIRECTIVE && p.name === 'slot'
}
```

有了该函数之后，我们就可以创建 `vif` 模块：

1. 创建 `packages/compiler-core/src/transforms/vIf.ts` 模块：

代码块

```
/**
 * transformIf === exitFns。内部保存了所有 v-if、v-else、else-if 的处理函数
 */
export const transformIf = createStructuralDirectiveTransform(
    /^(if|else|else-if)$/,
    (node, dir, context) => {
        return processIf(node, dir, context, (ifNode, branch, isRoot) => {
            // TODO: 目前无需处理兄弟节点情况
            let key = 0

            // 退出回调。当所有子节点都已完成时，完成codegenNode
            return () => {
                if (isRoot) {
                    ifNode.codegenNode = createCodegenNodeForBranch(branch, key, con
                } else {
                    // TODO: 非根
                }
            }
        })
    }
)
```

2. 构建 `processIf` 函数，为具体的 `if` 处理函数：

代码块

```
/**
 * v-if 的转化处理
 */
export function processIf(
    node,
    dir,
    context: TransformContext,
    processCodegen?: (node, branch, isRoot: boolean) => (() => void) | undefined
) {
    // 仅处理 v-if
    if (dir.name === 'if') {
        // 创建 branch 属性
        const branch = createIfBranch(node, dir)
        // 生成 if 指令节点，包含 branches
        const ifNode = {
            type: NodeTypes.IF,
            loc: node.loc,
            branches: [branch]
        }
        // 切换 currentVNode，即：当前处理节点为 ifNode
        context.replaceNode(ifNode)
        // 生成对应的 codegen 属性
        if (processCodegen) {
            return processCodegen(ifNode, branch, true)
        }
    }
}
```

3. 创建 `createIfBranch` 函数：

代码块

```
/**
 * 创建 if 指令的 branch 属性节点
```

```
function createIfBranch(node, dir) {
    return {
        type: NodeTypes.IF_BRANCH,
        loc: node.loc,
        condition: dir.exp,
        children: [node]
    }
}
```

4. 在 `packages/compiler-core/src/transform.ts` 中为 `context`，添加 `replaceNode` 函数：

代码块

```
/**
 * transform 上下文对象
 */
export interface TransformContext {
    ...
    /**
     * 替换节点
     */
    replaceNode(node): void
}

/**
 * 创建 transform 上下文
 */
export function createTransformContext(
    root,
    { nodeTransforms = [] }
): TransformContext {
    const context: TransformContext = {
        ...
        replaceNode(node) {
            context.parent!.children[context.childIndex] = context.currentNode = nod
        }
    }

    return context
}
```

5. 创建 `createCodegenNodeForBranch` 函数，为整个分支节点，添加 `codegen` 属性：

代码块

```
/**
 * 生成分支节点的 codegenNode
 */
function createCodegenNodeForBranch(
    branch,
    keyIndex: number,
    context: TransformContext
) {
    if (branch.condition) {
        return createConditionalExpression(
            branch.condition,
            createChildrenCodegenNode(branch, keyIndex),
            // 以注释的形式展示 v-if.
            createCallExpression(context.helper(CREATE_COMMENT), ['"v-if"', 'true'])
        )
    } else {
        return createChildrenCodegenNode(branch, keyIndex)
    }
}
```

6. 在 `packages/compiler-core/src/runtimeHelpers.ts` 中，增加 `CREATE_COMMENT`：

代码块

```
    ...
    [CREATE_COMMENT]: 'createCommentVNode'
}
```

7. 在 `packages/compiler-core/src/ast.ts` 中创建 `createCallExpression` 方法：

代码块

```
/**
 * 创建调用表达式的节点
 */
export function createCallExpression(callee, args) {
    return {
        type: NodeTypes.JS_CALL_EXPRESSION,
        loc: {},
        callee,
        arguments: args
    }
}
```

8. 在 `packages/compiler-core/src/ast.ts` 中创建 `createConditionalExpression` 方法：

代码块

```
/**
 * 创建条件表达式的节点
 */
export function createConditionalExpression(
    test,
    consequent,
    alternate,
    newline = true
) {
    return {
        type: NodeTypes.JS_CONDITIONAL_EXPRESSION,
        test,
        consequent,
        alternate,
        newline,
        loc: {}
    }
}
```

9. 最后创建创建 `createChildrenCodegenNode` 方法，用来处理子节点的 `codegen`：

代码块

```
/**
 * 创建指定子节点的 codegen 节点
 */
function createChildrenCodegenNode(branch, keyIndex: number) {
    const keyProperty = createObjectProperty(
        `key`,
        createSimpleExpression(`${keyIndex}`, false)
    )
    const { children } = branch
    const firstChild = children[0]

    const ret = firstChild.codegenNode
    const vnodeCall = getMemoedVNodeCall(ret)
    // 填充 props
    injectProp(vnodeCall, keyProperty)
    return ret
}
```

10. 在 `packages/compiler-core/src/ast.ts` 中创建 `createObjectProperty` 和 `createSimpleExpression` 方法：

代码块

```
/**
```

```
 */
export function createSimpleExpression(content, isStatic) {
    return {
        type: NodeTypes.SIMPLE_EXPRESSION,
        loc: {},
        content,
        isStatic
    }
}

/**
 * 创建对象属性节点
 */
export function createObjectProperty(key, value) {
    return {
        type: NodeTypes.JS_PROPERTY,
        loc: {},
        key: isString(key) ? createSimpleExpression(key, true) : key,
        value
    }
}
```

11. 在 `packages/compiler-core/src/utils.ts` 中创建 `getMemoedVNodeCall` 方法：

代码块

```
/**
 * 返回 vnode 节点
 */
export function getMemoedVNodeCall(node) {
    return node
}
```

12. 最后在 `packages/compiler-core/src/utils.ts` 中创建 `injectProp` 方法：

代码块

```
/**
 * 填充 props
 */
export function injectProp(node, prop) {
    let propsWithInjection
    let props =
        node.type === NodeTypes.VNODE_CALL ? node.props : node.arguments[2]

    if (props == null || isString(props)) {
        propsWithInjection = createObjectExpression([prop])
    }
    if (node.type === NodeTypes.VNODE_CALL) {
        node.props = propsWithInjection
    }
}
```

13. 该方法依赖 `createObjectExpression` ，所以直接创建 `createObjectExpression` 方法：

代码块

```
/**
 * 创建对象表达式节点
 */
export function createObjectExpression(properties) {
    return {
        type: NodeTypes.JS_OBJECT_EXPRESSION,
        loc: {},
        properties
    }
}
```

至此，我们完成了对应的 `VIF` 模块，接下来我们就只需要在 `transform` 的适当实际，触发该模块即可。

10：基于编辑器的指令(v-xx)处理：AST 解... 上一节 下一节 12：基于编辑器的指令(v-xx)处理：JavaScri...

我要提出意见反馈