Sunday • 更新于 2022-10-19

上一节 09：源码阅读：... 11：框架实现：... 下一节

10：框架实现：转化 JavaScript AST，构建深度优先的 AST 转化逻辑

明确好了 `transform` 的大致逻辑之后，这一小节我们就开始实现一下对应的代码，我们代码的逻辑实现我们分成两个小节来讲：

1. 深度优先排序
2. 完成具体的节点转化

这一小节，我们先来完成深度优先排序：

1. 在 `packages/compiler-core/src/compile.ts` 的 `baseCompile` 中，增加 `transform` 的方法触发：

代码块

```
export function baseCompile(template: string, options = {}) {
    const ast = baseParse(template)

    transform(
        ast,
        extend(options, {
            nodeTransforms: [transformElement, transformText]
        })
    )
    console.log(JSON.stringify(ast))

    return {}
}
```

2. 创建 `packages/compiler-core/src/transforms/transformElement.ts` 模块，导出 `transformElement` 方法：

代码块

```
/**
 * 对 element 节点的转化方法
 */
export const transformElement = (node, context) => {
    return function postTransformElement() {

    }
}
```

3. 创建 `packages/compiler-core/src/transforms/transformText.ts` 模块，导出 `transformText` 方法：

代码块

```
export const transformText = (node, context) => {
    if (
        node.type === NodeTypes.ROOT ||
        node.type === NodeTypes.ELEMENT ||
        node.type === NodeTypes.FOR ||
        node.type === NodeTypes.IF_BRANCH
    ) {
        return () => {
```

```
}

```

4. 创建 `packages/compiler-core/src/transform.ts` 模块，创建 `transform` 方法：

代码块

```
/**
 * 根据 AST 生成 JavaScript AST
 * @param root AST
 * @param options 配置对象
 */
export function transform(root, options) {
    // 创建 transform 上下文
    const context = createTransformContext(root, options)
  // 按照深度优先依次处理 node 节点转化
  traverseNode(root, context)
}
```

5. 创建 `createTransformContext` 生成上下文对象：

代码块

```
/**
 * transform 上下文对象
 */
export interface TransformContext {
    /**
     * AST 根节点
     */
    root
    /**
     * 每次转化时记录的父节点
     */
    parent: ParentNode | null
    /**
     * 每次转化时记录的子节点索引
     */
    childIndex: number
    /**
     * 当前处理的节点
     */
    currentNode
    /**
     * 协助创建 JavaScript AST 属性 helpers，该属性是一个 Map，key 值为 Symbol(方法名),
     */
    helpers: Map<symbol, number>
    helper<T extends symbol>(name: T): T
    /**
     * 转化方法集合
     */
    nodeTransforms: any[]
}

/**
 * 创建 transform 上下文
 */
export function createTransformContext(
    root,
    { nodeTransforms = [] }
): TransformContext {
    const context: TransformContext = {
        // options
        nodeTransforms,

        // state
        root,
        helpers: new Map(),
        currentNode: root,
        parent: null,
        childIndex: 0,
```

```
        // methods
        helper(name) {
            const count = context.helpers.get(name) || 0
            context.helpers.set(name, count + 1)
            return name
        }
    }

    return context
}
```

6. 创建 `traverseNode` 方法：

代码块

```
/**
 * 遍历转化节点，转化的过程一定要是深度优先的（即：孙 -> 子 -> 父），因为当前节点的状态往往
 * 转化的过程分为两个阶段：
 * 1. 进入阶段：存储所有节点的转化函数到 exitFns 中
 * 2. 退出阶段：执行 exitFns 中缓存的转化函数，且一定是倒叙的。因为只有这样才能保证整个处理
 */
export function traverseNode(node, context: TransformContext) {
    // 通过上下文记录当前正在处理的 node 节点
    context.currentNode = node
    // 获取当前所有 node 节点的 transform 方法
    const { nodeTransforms } = context
    // 存储转化函数的数组
    const exitFns: any = []
    // 循环获取节点的 transform 方法，缓存到 exitFns 中
    for (let i = 0; i < nodeTransforms.length; i++) {
        const onExit = nodeTransforms[i](node, context)
        if (onExit) {
            exitFns.push(onExit)
        }
    }

    // 继续转化子节点
    switch (node.type) {
        case NodeTypes.ELEMENT:
        case NodeTypes.ROOT:
            traverseChildren(node, context)
            break
    }

    // 在退出时执行 transform
    context.currentNode = node
    let i = exitFns.length
    while (i--) {
        exitFns[i]()
    }
}

/**
 * 循环处理子节点
 */
export function traverseChildren(parent, context: TransformContext) {
    parent.children.forEach((node, index) => {
        context.parent = parent
        context.childIndex = index
        traverseNode(node, context)
    })
}
```

那么至此，一个按照深度优先依次处理 node 节点转化的逻辑就已经完成