Vue3 源码解读

来源：https://wjchumble.github.io/explain-vue3

准备工作

Monorepo

> 原文：https://wjchumble.github.io/explain-vue3/chapter1/

在「Vue3」，它采用了「Monorepo」的方式来管理项目的代码。那么，什么是「Monorepo」？我们先来看看维基百科上对「Monorepo」的介绍：

———— In revision control systems, a monorepo is a software development strategy where code for many projects is stored in the same repository.

简单理解，「Monorepo」**指一种将多个项目放到一个仓库的一种管理项目的策略**。当然，这只是概念上的理解。而对于实际开发中的场景，「Monorepo」的使用通常是通过 `yarn` 的 `workspaces` 工作空间，又或者是 `lerna` 这种高度封装的第三方工具库来实现。使用「Monorepo」的方式来管理项目会给我们带来以下这些好处：

- 只需要一个仓库，就可以便捷地管理多个项目
- 可以管理不同项目中的相同第三方依赖，做到依赖的同步更新
- 可以使用其他项目中的代码，清晰地建立起项目间的依赖关系

而「Vue3」正是采用的 `yarn` 的 `workspaces` 工作空间的方式管理整个项目，而 `workspaces` 的特点就是在 `package.json` 中会有这么两句不同于普通项目的声明：

```
{
    "private": true,
    "workspaces": [
        "packages/*"
```

```
    ]
}
```

其中 `"private": true` 的作用是保证了工作区的安全，避免被其他引用， `"workspaces"` 则是用来声明工作区所包含的项目的位置，很显然它可以声明多个，而 `packages/*` 指的是 `packages` 文件夹下的所有项目。并且，「Vue3」中对工作区的声明也是 `pacakges/*`，所以它的目录结构会是这样：

```
...
|── packages
    |──── compiler-core
    |──── compiler-dom
    |──── compiler-sfc
    |──── compiler-ssr
    |──── reactivity
    |──── runime-core
    |──── runime-dom
    |──── runime-test
    |──── server-renderer
    |──── shared
    |──── size-check
    |──── template-explorer
    |──── vue
    global.dt.s
package.json
```

> 这里我只展示了 `packages` 文件目录和 `package.json`，至于其他目录有兴趣的同学可以自行了解。

可以看到，`packages` 文件目录下根据「Vue3」实现所需要的能力划分了不同的项目，例如 `reactivity` 文件目录下就是和 `reactivity` API 相关的代码，并且它的内部的结构会是这样：

```
|── __tests__                    ## 测试用例
|── src                          ## reactive API 实现相关
api.extractor.json
index.js
LICENSE
package.json                     ## reactive API 实现相关
README.md
```

在 `reactivity` 项目文件的内部也同样有 `package.json` 文件，也就是如我们上面所说的，`packages` 文件目录下的文件都各自对应着每一个单独的项目。所以，每一个项目中的 `package.json` 就对应着改项目对应的依赖、入口、打包的一些配置等等。

而「Vue3」使用「Monorepo」的方式管理项目的好处就是我们可以单独使用它的一些 API 的能力，而不是只能在「Vue」项目中使用它。很典型的例子就是，我们可以通过 `npm i @vue/reactivity` 单独安装 `reactivity` API 对应的 `npm` 包，从而在其他地方使用 `reactivity` API 来实现观察者模式。

> 当然，使用「Monorepo」还需要思考诸多其他问题，例如增量编译、多任务编译等等，有兴趣同学可以自行去了解。

总结

那么，在简单介绍完「Vue3」是以「Monorepo」的方式管理项目后。我想，大家心中都已明了，如果我们要去了解「Vue3」怎么去实现模板编译、runtime + compiler 的巧妙结合、Virtual DOM 的实现等等原理，我们就可以从 `packages` 文件目录下的各个文件开始着手来研究它们的实现。

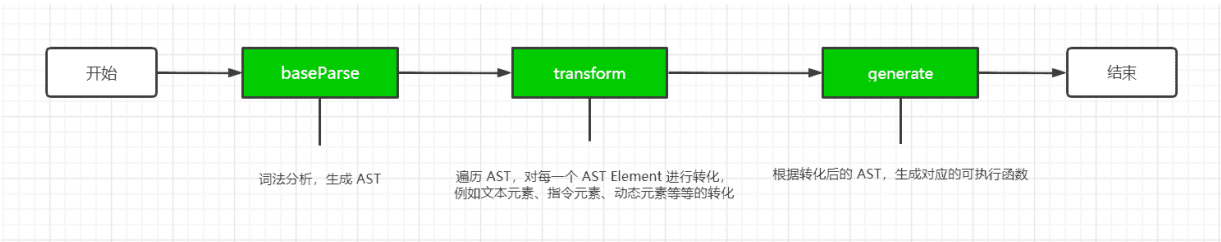
模板编译

基本介绍

> 原文：https://wjchumble.github.io/explain-vue3/chapter2/

在 Vue3 中编译是由 `compiler-core` 这个 `package` 完成的，其顾名思义即核心的编译，它会做这么三件事：

- **baseParse**，对组件 `template` 进行词法分析，生成对应的抽象语法树 AST。
- **transfrom**（转化）AST，针对每一个 AST Element，进行不同的 transform 处理，例如 `v-on`、`slot`、`v-if`、纯文本元素等等。
- **generate**，根据转化后的 AST 来生成对应的可执行函数。



而这三个过程主要是由 `baseCompiler` 负责来完成，它对应的伪代码会是这样：

baseCompiler 函数

```
export function baseCompile(
  template: string | RootNode,
  options: CompilerOptions = {}
): CodegenResult {
  ...
  const ast = isString(template) ? baseParse(templ
  ...
  transform(
    ast,
    extend({}, options, {....})
```

```
  )

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

假设，我们此时有这么一个栗子，它的 `template` 会是这样：

```
<div>
  <div>hi vue3</div>
  <div>{{msg}}</div>
</div>
```

其中 `msg` 是一个插值，对应的值为 `hello vue3`。而在 `compiler-core` 时，它的核心方法是 `baseCompiler`，它会通过调用 `baseParse` 函数来将这个 `template` 解析成 AST。

那么，我们这个栗子，它经过 `baseParse` 处理后生成对应的 AST 会是这样：

```
{
  cached: 0
  children: [{…}]
  codegenNode: undefined
  components: []
  directives: []
  helpers: []
  hoists: []
  imports: []
  loc: {start: {…}, end: {…}, source: "<div><div>h
  temps: 0
```

```
  type: 0
}
```

> 这里先不展开 children 中的 AST Element，后面会一一涉及。

如果，了解过「Vue 2.x」编译过程的同学应该对于上面这颗 AST 的大部分属性不会陌生。 AST 的本质是通过用对象来描述 DSL（特殊领域语言），例如：

- `children` 中存放的就是最外层 `div` 的子代。
- `loc` 则用来描述这个 AST Element 在整个字符串（template）中的位置信息。
- `type` 则是用于描述这个元素的类型（例如 5 为插值、2 为文本）等等。

我想大家可能会有疑问的就是 `codegenNode`、`hoists` 这两个属性。而这两个属性也是「Vue 3」针对**更新性能**问题所添加的两个属性。对于前者 `codegenNode` 是用于描述该节点在 `generate` 的一些表现。对于后者 `hoists` 是用于**存储需要静态提升的节点**。

那么，对于 `codegenNode` 它又是怎么来的？从上面的 AST，可以看到它的 `codegenNode` 是 `undefined`，也就是在 `parse` 阶段，并不会处理生成 `codegen`。

而真正处理生成 AST Element 对应的 `codegenNode` 是在 `transform` 阶段完成。在这个阶段，它会执行很多 `transform` 函数，对于我们这个栗子，会命中两个比较特殊的 `transform` 函数，它分别是：`transformText`、`transformElement`。

> 原文：https://wjchumble.github.io/explain-vue3/chapter2/baseParse

baseParse

`baseParse` 顾名思义起着**解析**的作用，它的表现和「Vue2.x」的 `parse` 相同，都是解析模板 `tempalte` 生成**原始 AST**。

假设，此时我们有一个这样的模板 `template`：

```
<div>
  <div>hi vue3</div>
  <div>{{ msg }}</div>
</div>
```

那么，它在经过 `baseParse` 处理后生成的 AST 看起来会是这样：

```
{
  cached: 0,
  children: [{…}],
  codegenNode: undefined,
  components: [],
  directives: [],
  helpers: [],
  hoists: [],
  imports: [],
  loc: {start: {…}, end: {…}, source: "<div><div>h
  temps: 0,
  type: 0
}
```

如果，了解过「Vue2.x」编译过程的同学应该对于上面这颗 `AST` 的大部分属性不会陌生。`AST` 的本质是通过用对象来描述「DSL」（特殊领域语言），例如：

- `children` 中存放的就是最外层 `div` 的后代。
- `loc` 则用来描述这个 AST Element 在整个字符串（`template`）中的位置信息。
- `type` 则是用于描述这个元素的类型（例如 5 为插值、2 为文本）等等。

并且，可以看到的是不同于「Vue2.x」的 AST，这里我们多了诸如 `helpers`、`codegenNode`、`hoists` 等属性。而，这些属性会在 `transform` 阶段进行相应地赋值，进而帮助 `generate` 阶段生成**更优的**可执行代码。

静态节点 transform

> 原文：https://wjchumble.github.io/explain-vue3/chapter2/transform

transform

熟悉 Vue 2.x 版本源码的同学应该都知道它的 `compile` 阶段是没有 `transform` 过程的处理。而 `transform` 恰恰是整个 Vue 3 提高 `VNode` 更新性能实现的基础。因为，在这个阶段，会对 `baseCompiler` 后生成的 AST Element 打上优化标识 `patchFlag`，以及 `isBlock` 的判断。

> 实际上 Vue 3 的 `transfrom` 并不是无米之炊，它本质上是 Vue 2.x `compiler` 阶段的 `optimize` 的升级版。

这里我们将对 AST Elment 的 `transform` 分为两类：

- 静态节点 `transform` 应用，即节点不含有插值、指令、props、动态样式的绑定等。
- 动态节点 `transform` 应用，即节点含有插值、指令、props、动态样式的绑定等。

那么，首先是静态节点 `transform` 应用。对于上面我们说到的这个栗子，静态节点就是 `<div>hi vue3</div>`这部分。而它在没有进行 `transformText` 之前，它对应的 AST 会是这样：

```
{
  children: [{
    content: "hi vue3"
    loc: {start: {…}, end: {…}, source: "hi vue3"}
    type: 2
```

```
  }]
  codegenNode: undefined
  isSelfClosing: false
  loc: {start: {…}, end: {…}, source: "<div>hi vue
  ns: 0
  props: []
  tag: "div"
  tagType: 0
  type: 1
}
```

可以看出，此时它的 `codegenNode` 是 `undefined`。而在 `transform` 阶段则会根据 AST 递归应用对应的 `plugin`，然后，创建对应 AST Element 的 `codegen` 对象。所以，此时我们会命中 `transformElement` 和 `transformText` 的逻辑。

transformText

`transformText` 顾名思义，它和**文本**相关。很显然，我们此时 AST Element 所属的类型就是 Text。那么，我们先来看一下 `transformText` 函数对应的伪代码：

```
export const transformText: NodeTransform = (node,
  if (
    node.type === NodeTypes.ROOT ||
    node.type === NodeTypes.ELEMENT ||
    node.type === NodeTypes.FOR ||
    node.type === NodeTypes.IF_BRANCH
  ) {
    return () => {
      const children = node.children
      let currentContainer: CompoundExpressionNode
      let hasText = false

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

```
        const child = children[i]
        if (isText(child)) {
          hasText = true
          ...
        }
      }
      if (
        !hasText ||
        (children.length === 1 &&
          (node.type === NodeTypes.ROOT ||
            (node.type === NodeTypes.ELEMENT &&
              node.tagType === ElementTypes.ELEMEN
      ) { // {2}
        return
      }
      ...
    }
  }
}
```

可以看到，这里我们会命中 {2} 逻辑，即如果对于节点含有单一文本 `transformText` 并不需要进行额外的处理。该节点仍然和 Vue 2.x 版本一样，会交给 `runtime` 时的 `render` 函数处理。

> 至于 `transfromText` 真正发挥作用的场景是当存在 `<div>ab {a} {b}</div>` 情况时，它需要将两者放在一个单独的 AST Element (Compound Expression) 下。

transformElement

`transformElement` 是一个所有 AST Element 都会被执行的一个 `plugin`，它的核心是为 AST Element 生成最基础的

`codegen`属性。例如标识出对应 `patchFlag`，从而为生成 `VNode` 提供依据，即 `dynamicChildren`。

而对于静态节点，同样只是起到一个初始化它的 `codegenNode` 属性的作用。并且，从上面介绍的 `patchFlag` 的类型，我们可以知道它的 `patchFlag` 为默认值 `0`。所以，它的 `codegenNode` 属性值看起来会是这样：

```
{
  children: {
    content: "hi vue3"
    loc: {start: {…}, end: {…}, source: "hi vue3"}
    type: 2
  }
  directives: undefined
  disableTracking: false
  dynamicProps: undefined
  isBlock: false
  loc: {start: {…}, end: {…}, source: "<div>hi vue
  patchFlag: undefined
  props: undefined
  tag: ""div""
  type: 13
}
```

动态节点 transform

接下来是动态节点 `transform` 应用。这里，我们的动态节点是 `<div></div>`。它在 `baseParse` 后对应的 AST 会是这样：

```
{
  children: [
    {
```

```
      content: {type: 4, isStatic: false, isConsta
      loc: {start: {…}, end: {…}, source: "{{msg}}
      type: 5
    }
  ],
  codegenNode: undefined,
  isSelfClosing: false,
  loc: {start: {…}, end: {…}, source: "<div>{{msg}
  ns: 0,
  props: [],
  tag: "div",
  tagType: 0,
  type: 1
}
```

很显然 也是文本，所以也会命中和 `hi vue3` 一样的 `transformText` 函数的逻辑。

> 这里就不对 `transformText` 做展开，因为表现和 `hi vue3` 一样。

transformElements

此时，对于插值文本，`transfromElements` 的价值就会体现出来了。而针对存在单一节点的插值文本，它会两件事：

- 标识 `patchFlag` 为 `1 /* TEXT */`，即动态的文本内容。
- 将插值文本对应的 AST Element 赋值给 `VNodeChildren`。

具体在源码中的表现会是这样（伪代码）：

```
    ...
    if (node.children.length === 1 && vnodeTag !==
        const child = node.children[0]
        const type = child.type
```

```
        // check for dynamic text children
        const hasDynamicTextChild =
          type === NodeTypes.INTERPOLATION ||
          type === NodeTypes.COMPOUND_EXPRESSION
        if (hasDynamicTextChild && !getStaticType(
          patchFlag |= PatchFlags.TEXT
        }
        if (hasDynamicTextChild || type === 2 /* T
            vnodeChildren = child;
        }
    }
    if (patchFlag !== 0) {
      if (__DEV__) {
        ...
        // bitwise flags
        const flagNames = Object.keys(PatchFlagNam
          .map(Number)
          .filter(n => n > 0 && patchFlag & n)
          .map(n => PatchFlagNames[n])
          .join(`, `)
        vnodePatchFlag = patchFlag + ` /* ${flagNa
        ...
    }
    ...
    node.codegenNode = createVNodeCall(
      context,
      vnodeTag,
      vnodeProps,
      vnodeChildren,
      vnodePatchFlag,
      vnodeDynamicProps,
      vnodeDirectives,
      !!shouldUseBlock,
      false /* disableTracking */,
      node.loc
    )
```

可以看到，处理后的 `vnodePatchFlag` 和 `vnodeChildren` 是作为参数传入 `createVNodeCall`，而 `createVNode` 最终会将这些参数转化为 AST Element 上属性的值，例如 `children`、`patchFlag`。所以，`transformElement` 处理后，其生成对应的 `codegenNode` 属性值会是这样：

```
{
  children: {
    type: 4,
    isStatic: false,
    isConstant: false,
    content: "msg",
    loc: {…}
  },
  directives: undefined,
  dynamicProps: undefined,
  isBlock: false,
  isForBlock: false,
  loc: {
    start: {…},
    end: {…},
    source: "<div>{{msg}}</div>"
  },
  patchFlag: "1 /* TEXT */",
  props: undefined,
  tag: ""div"",
  type: 13
}
```

CodegenContext

> 原文：https://wjchumble.github.io/explain-vue3/chapter2/generate

generate

`generate` 是 `compile` 阶段的最后一步，它的作用是将 `transform` 转换后的 AST 生成对应的**可执行代码**，从而在之后 Runtime 的 Render 阶段时，就可以通过可执行代码生成对应的 VNode Tree，然后最终映射为真实的 DOM Tree 在页面上。

同样地，这一阶段在「Vue2.x」也是由 `generate` 函数完成，它会生成是诸如 `_l`、`_c` 之类的函数，这本质上是对 `_createElement` 函数的封装。而相比较「Vue2.x」版本的 `generate`，「Vue3」改变了很多，其 `generate` 函数对于的伪代码会是这样：

```
export function generate(
  ast: RootNode,
  options: CodegenOptions & {
    onContextCreated?: (context: CodegenContext) =
  } = {}
): CodegenResult {
  const context = createCodegenContext(ast, option
  if (options.onContextCreated) options.onContextC
  const {
    mode,
    push,
    prefixIdentifiers,
    indent,
    deindent,
```

```
    newline,
    scopeId,
    ssr
  } = context
  ...
  genFunctionPreamble(ast, context)
  ...

  if (!ssr) {
    ...
    push(`function render(_ctx, _cache${optimizeSo
  }
  ....

  return {
    ast,
    code: context.code,
    // SourceMapGenerator does have toJSON() metho
    map: context.map ? (context.map as any).toJSON
  }
}
```

所以，接下来，我们就来**一睹**带有静态节点对应的 AST 生成的可执行代码的过程会是怎样。

从上面 `generate` 函数的伪代码可以看到，在函数的开始调用了 `createCodegenContext` 为当前 AST 生成了一个 `context`。在整个 `generate` 函数的执行过程**都依托**于一个 `CodegenContext` **生成代码上下文**（对象）的能力，它是通过 `createCodegenContext` 函数生成。而 `CodegenContext` 的接口定义会是这样：

```
interface CodegenContext
  extends Omit<Required<CodegenOptions>, 'bindingM
```

```
  source: string
  code: string
  line: number
  column: number
  offset: number
  indentLevel: number
  pure: boolean
  map?: SourceMapGenerator
  helper(key: symbol): string
  push(code: string, node?: CodegenNode): void
  indent(): void
  deindent(withoutNewLine?: boolean): void
  newline(): void
}
```

可以看到 `CodegenContext` 对象中有诸如 `push`、`indent`、`newline` 之类的方法。而它们的作用是在根据 AST 来生成代码时用来**实现换行**、**添加代码**、**缩进**等功能。从而，最终形成一个个可执行代码，即我们所认知的 `render` 函数，并且，它会作为 `CodegenContext` 的 `code` 属性的值返回。

下面，我们就来看下静态节点的可执行代码生成的核心，它被称为 `Preamble` 前导。

genFunctionPreamble

整个静态提升的可执行代码生成就是在 `genFunctionPreamble` 函数部分完成的。并且，大家仔细**斟酌**一番静态提升的字眼，静态二字我们可以不看，但是**提升二字**，直抒本意地表达出它（静态节点）被**提高了**。

为什么说是提高了？因为在源码中的体现，确实是被提高了。在前面的 `generate` 函数，我们可以看到

`genFunctionPreamble` 是先于 `render` 函数加入 `context.code` 中，所以，在 Runtime 阶段的 Render，它会先于 `render` 函数执行。

`geneFunctionPreamble` 函数（伪代码）：

```
function genFunctionPreamble(ast: RootNode, contex
  const {
    ssr,
    prefixIdentifiers,
    push,
    newline,
    runtimeModuleName,
    runtimeGlobalName
  } = context
  ...
  const aliasHelper = (s: symbol) => `${helperName
  if (ast.helpers.length > 0) {
    ...
    if (ast.hoists.length) {
      const staticHelpers = [
        CREATE_VNODE,
        CREATE_COMMENT,
        CREATE_TEXT,
        CREATE_STATIC
       ]
        .filter(helper => ast.helpers.includes(hel
        .map(aliasHelper)
        .join(', ')
      push(`const { ${staticHelpers} } = _Vue\n`)
    }
  }
  ...
  genHoists(ast.hoists, context)
  newline()
```

```
  push(`return `)
}
```

可以看到，这里会对前面我们在 `transform` 函数提及的 `hoists` 属性的长度进行判断。显然，对于前面说的这个栗子，它的 `ast.hoists.length` 长度是大于 0 的。所以，这里就会根据 `hoists` 中的 AST 生成对应的可执行代码。因此，到这里，生成的可执行代码会是这样：

```
const _Vue = Vue;
const { createVNode: _createVNode } = _Vue;
// 静态提升部分
const _hoisted_1 = _createVNode("div", null, "hi v
// render 函数会在这下面
export function render() => render(_ctx, _cache, $
        return (_openBlock(), _createElementBlock(
    _hoisted_1,
    _createElementVNode("div", null, _toDisplayStr
  ], 64 /* STABLE_FRAGMENT */)
        )
}
```

总结

组件创建过程

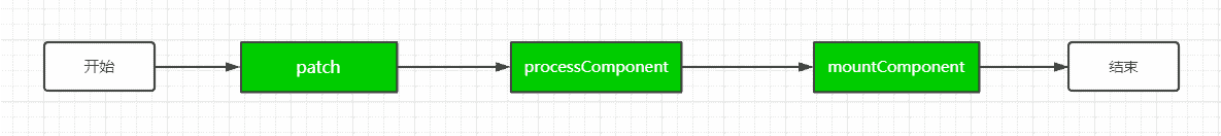
基本介绍

> 原文：https://wjchumble.github.io/explain-vue3/chapter3/

在「Vue3」中，创建一个组件实例由 `createApp` 「API」完成。创建完一个组件实例，我们需要调用 `mount()` 方法将组件实例挂载到页面中：

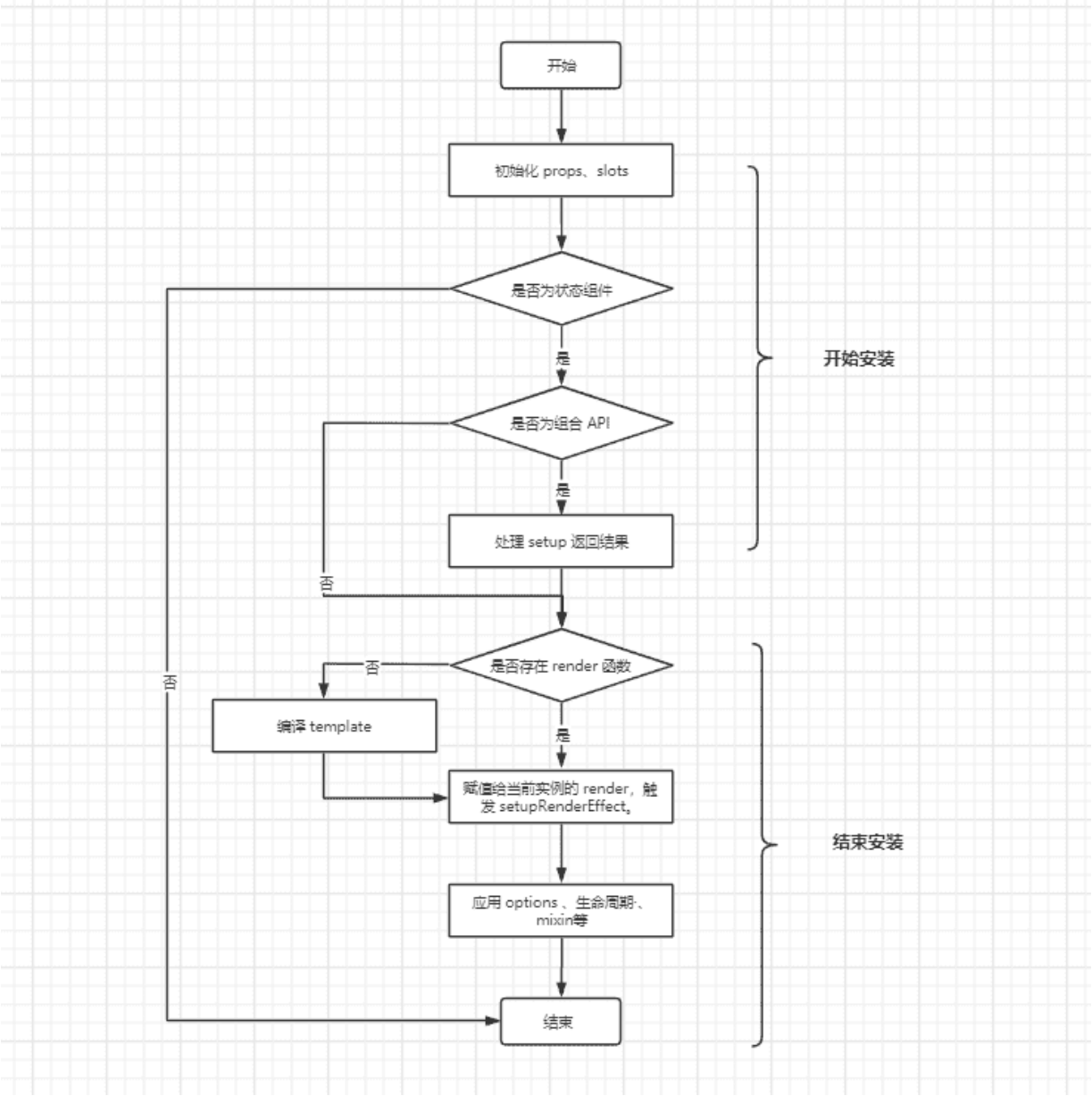
```
createApp({
    ...
}).mount("#app");
```

在源码中整个组件的创建过程:



`mountComponent()` 实现的核心是 `setupComponent()`，它可以分为**两个过程**：

- 开始安装，它会初始化 `props`、`slots`、调用 `setup()`、验证组件和指令的合理性。
- 结束安装，它会初始化 `computed`、`data`、`watch`、`mixin` 和生命周期等等。



那么，接下来我们来详细地分析一下这两个过程。

setupStatefulComponent

> 原文：https://wjchumble.github.io/explain-vue3/chapter3/setupComponent

setupComponent

`setupComponent()` 的定义：

```
// packages/runtime-core/src/component.ts
function setupComponent(instance: ComponentInterna
  isInSSRComponentSetup = isSSR;

  const { props, children, shapeFlag } = instance.
  const isStateful = shapeFlag & ShapeFlags.STATEF
  initProps(instance, props, isStateful, isSSR); /
  initSlots(instance, children); // {C}

  const setupResult = isStateful
    ? setupStatefulComponent(instance, isSSR)
    : undefined; // {D}
  isInSSRComponentSetup = false;
  return setupResult;
}
```

抛开 `SSR` 的逻辑，B 行和 C 行会先初始化组件的 `props` 和 `slots`。然后，在 A 行判断 `shapeFlag` 为 `true` 时，调用 `setupStatefulComponent()`。

> 这里又用到了 `shapeFlag`，所以需要强调的是 `shapeFlag` 和 `patchFlag` 具有一样的地位（重要性）。

而 `setupStatefulComponent()` 则会处理组合 `Composition API`，即调用 `setup()`。

`setupStatefulComponent()` 定义（伪代码）：

```
// packages/runtime-core/src/component.ts
setupStatefulComponent(
  instance: ComponentInternalInstance,
  isSSR: boolean
) {
  const Component = instance.type as ComponentOpti
  // {A} 验证逻辑
  ...
  instance.proxy = new Proxy(instance.ctx, PublicI
  ...
  const { setup } = Component
  if (setup) {
    const setupContext = (instance.setupContext =
      setup.length > 1 ? createSetupContext(instan

    currentInstance = instance // {B}
    pauseTracking() // {C}
    const setupResult = callWithErrorHandling(
      setup,
      instance,
      ErrorCodes.SETUP_FUNCTION,
      [__DEV__ ? shallowReadonly(instance.props) :
    ) // {D}
    resetTracking() // {E}
    currentInstance = null

    if (isPromise(setupResult)) {
      ...
    } else {
      handleSetupResult(instance, setupResult, isS
```

```
    }
  } else {
    finishComponentSetup(instance, isSSR)
  }
}
```

首先，在 B 行会给当前实例 `currentInstance` 赋值为此时的组件实例 `instance`，在回收 `currentInstance` 之前，我们会做两个操作**暂停依赖收集**、**恢复依赖收集**：

暂停依赖收集 `pauseTracking()`：

```
// packages/reactivity/src/effect.ts
function pauseTracking() {
  trackStack.push(shouldTrack);
  shouldTrack = false;
}
```

恢复依赖收集 `resetTracking()`：

```
// packages/reactivity/src/effect.ts
resetTracking() {
  const last = trackStack.pop()
  shouldTrack = last === undefined ? true : last
}
```

本质上这两个步骤是通过改变 `shouldTrack` 的值为 `true` 或 `false` 来控制此时是否进行依赖收集。之所以，`shouldTrack` 可以控制是否进行依赖收集，是因为在 `track` 的执行开始有这么一段代码：

```
// packages/reactivity/src/effect.ts
function track(target: object, type: TrackOpTypes,
  if (!shouldTrack || activeEffect === undefined)
```

```
    return
  }
  ...
}
```

那么，我们就会提出疑问为什么这个时候需要**暂停依赖收**？这里，我们回到 D 行：

```
const setupResult = callWithErrorHandling(
  setup,
  instance,
  ErrorCodes.SETUP_FUNCTION,
  [__DEV__ ? shallowReadonly(instance.props) : ins
); // {D}
```

在 `DEV` 环境下，我们需要通过 `shallowReadonly(instance.props)` 创建一个基于组件 `props` 的拷贝对象 `Proxy`，而 `props` 本质上是**响应式地**，这个时候会触发它的 `track` 逻辑，即依赖收集，明显这并**不是开发中实际需要**的订阅对象，所以，此时要暂停 `props` 的依赖收集，**过滤不必要的订阅**。

> 相比较，「Vue2.x」泛滥的订阅关系而言，这里不得不给「Vue3」对订阅关系处理的严谨思维点赞！

通常，我们 `setup()` 返回的是一个 `Object`，所以会命中 F 行的逻辑：

```
handleSetupResult(instance, setupResult, isSSR);
```

handleSetupResult

`handleSetupResult()` 定义：

```
// packages/runtime-core/src/component.ts
function handleSetupResult(
  instance: ComponentInternalInstance,
  setupResult: unknown,
  isSSR: boolean
) {
  if (isFunction(setupResult)) {
    instance.render = setupResult as InternalRende
  } else if (isObject(setupResult)) {
    if (__DEV__ && isVNode(setupResult)) {
      warn(
        `setup() should not return VNodes directly
          `return a render function instead.`
      )
    }
    instance.setupState = proxyRefs(setupResult)
    if (__DEV__) {
      exposeSetupStateOnRenderContext(instance)
    }
  } else if (__DEV__ && setupResult !== undefined)
    warn(
      `setup() should return an object. Received:
        setupResult === null ? 'null' : typeof set
      }`
    )
  }
  finishComponentSetup(instance, isSSR)
}
```

`handleSetupResult()` 的分支逻辑较为简单，主要是验证 `setup()` 返回的结果，以下两种情况都是**不合法的**：

- `setup()` 返回的值是 `render()` 的执行结果，即 `VNode`。

- `setup()` 返回的值是 `null`、`undefined`或者其他非对象类型。

总结

到此，组件的开始安装过程就结束了。我们再来回顾一下这个过程会做的几件事，初始化 `props`、`slot`以及处理 `setup()` 返回的结果，期间还涉及到一个暂停依赖收集的微妙处理。

需要注意的是，此时组件并**没有开始创建**，因此我们称之为这个过程为**安装**。并且，这也是为什么官方文档会这么介绍 `setup()`：

> 一个组件选项，**在创建组件之前执行**，一旦 props 被解析，并作为组合 API 的入口点

applyOptions

> 原文：https://wjchumble.github.io/explain-vue3/chapter3/finishComponentSetup

finishComponentSetup

`finishComponentSetup()` 定义（伪代码）：

```
// packages/runtime-core/src/component.ts
function finishComponentSetup(
  instance: ComponentInternalInstance,
  isSSR: boolean
) {
  const Component = instance.type as ComponentOpti
  ...
  if (!instance.render) { // {A}
    if (compile && Component.template && !Componen
      ...
      Component.render = compile(Component.templat
        isCustomElement: instance.appContext.confi
        delimiters: Component.delimiters
      })
      ...
    }

    instance.render = (Component.render || NOOP) a
    if (instance.render._rc) {
      instance.withProxy = new Proxy(
        instance.ctx,
        RuntimeCompiledPublicInstanceProxyHandlers
      )
    }
  }
```

```
  if (__FEATURE_OPTIONS_API__) { // {C}
    currentInstance = instance
    applyOptions(instance, Component)
    currentInstance = null
  }
  ...
}
```

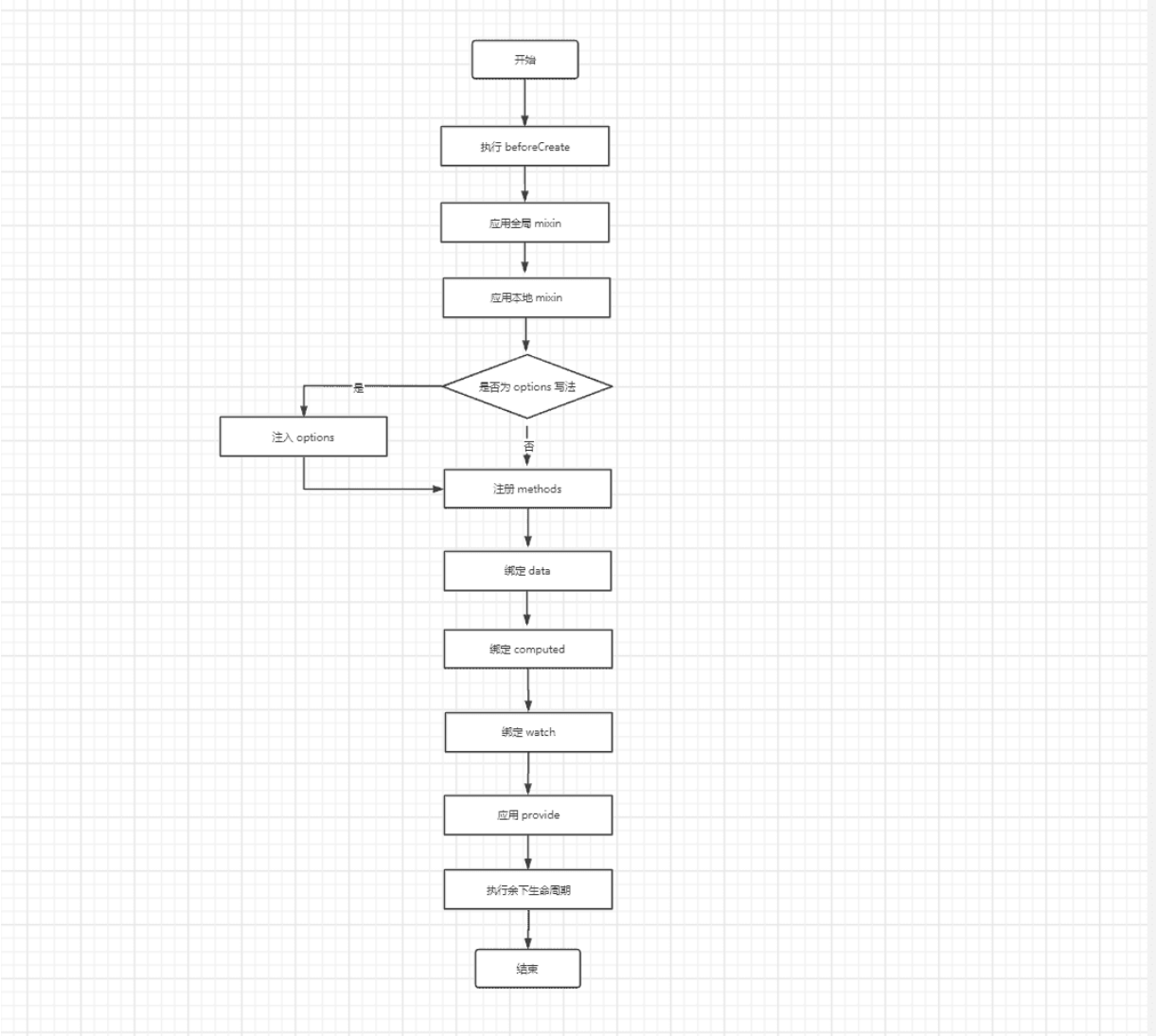
整体上 `finishComponentSetup()` 可以分为三个核心逻辑：

- 绑定 `render` 函数到当前实例 `instance` 上（行 A），这会两种情况，一是手写 `render` 函数，二是模板 `template` 写法，它会调用 `compile` 编译模板生成 `render` 函数。
- 为模板 `template` 生成的 `render` 函数（行 B），单独使用一个不同的 `has` 陷阱。因为，编译生成的 `render` 函数是会存在 `withBlock` 之类的优化，以及它会有一个全局的白名单来实现避免进入 `has` 陷阱。
- 应用 `options`（行 C），即对应的 `computed`、`watch`、`lifecycle` 等等。

`applyOptions()` 定义：

```
// packages/runtime-core/src/componentOptions.ts
function applyOptions(
  instance: ComponentInternalInstance,
  options: ComponentOptions,
  deferredData: DataFn[] = [],
  deferredWatch: ComponentWatchOptions[] = [],
  asMixin: boolean = false
) {
  ...
}
```

由于，`applyOptions()` 涉及的代码较多，我们先不看代码，看一下整体的流程：



`applyOptions()` 的流程并不复杂，但是从流程中我们总结出**两点**平常开发中忌讳的点：

- 不要在 `beforeCreate` 中访问 `mixin` 相关变量。
- 由于本地 `mixin` 后于全局 `mixin` 执行，所以在一些变量命名重复的场景，我们需要确认要使用的是全局 `mixin` 的这个变量还是本地的 `mixin`。

> 对于 `mixin` 重名时选择本地还是全局的处理，有兴趣的同学可以去官方文档了解。

我们再从代码层面看整个流程，这里分析几点常关注的属性是怎么初始化的：

注册事件（methods）

```
if (methods) {
  for (const key in methods) {
    const methodHandler = (methods as MethodOption
    if (isFunction(methodHandler)) {
      ctx[key] = methodHandler.bind(publicThis) //
      if (__DEV__) {
        checkDuplicateProperties!(OptionTypes.METH
      }
    } else if (__DEV__) {
      warn(
        `Method "${key}" has type "${typeof method
          `Did you reference the function correctl
      )
    }
  }
}
```

事件的注册，主要就是遍历已经处理好的 `methods` 属性，然后在当前上下文 `ctx` 中绑定对应事件名的属性 `key` 的事件 `methodHandler`（行 A）。并且，**在开发环境下**会对当前上下文属性的唯一性进行判断。

绑定计算属性（computed）

```
if (computedOptions) {
    for (const key in computedOptions) {
```

```
      const opt = (computedOptions as ComputedOpti
      const get = isFunction(opt)
        ? opt.bind(publicThis, publicThis)
        : isFunction(opt.get)
          ? opt.get.bind(publicThis, publicThis)
          : NOOP // {A}
      if (__DEV__ && get === NOOP) {
        warn(`Computed property "${key}" has no ge
      }
      const set =
        !isFunction(opt) && isFunction(opt.set)
          ? opt.set.bind(publicThis)
          : __DEV__
            ? () => {
                warn(
                  `Write operation failed: compute
                )
              }
            : NOOP // {B}
      const c = computed({
        get,
        set
      }) // {C}
      Object.defineProperty(ctx, key, {
        enumerable: true,
        configurable: true,
        get: () => c.value,
        set: v => (c.value = v)
      }) {D}
      if (__DEV__) {
        checkDuplicateProperties!(OptionTypes.COMP
      }
    }
  }
```

绑定计算属性主要是遍历构建好的 `computedOptions`，然后提取每一个计算属性 `key` 对应的 `get` 和 `set`（行 A），也是我们熟悉的对于 `get` 是**强校验**，即计算属性**必须要有 `get`，可以没有 `set`**，如果没有 `set`（行 B），此时它的 `set` 为：

```
() => {
  warn(`Write operation failed: computed property
};
```

> 所以，这也是为什么我们修改一个没有定义 `set` 的计算属性时会提示这样的错误。

然后，在 C 行会调用 `computed` 注册该计算属性，即 `effect` 的注册。最后，将该计算属性通过 `Object.defineProperty` 代理到当前上下文 `ctx` 中（行 D），保证通过 `this.computedAttrName` 可以获取到该计算属性。

生命周期处理

生命周期的处理比较特殊的是 `beforeCreate`，它是优于 `mixin`、`data`、`watch`、`computed` 先处理：

```
if (!asMixin) {
  callSyncHook("beforeCreate", options, publicThis
  applyMixins(instance, globalMixins, deferredData
}
```

至于其余的生命周期是在最后处理，即它们可以正常地访问实例上的属性（伪代码）：

```
if (lifecycle) {
  onBeforeMount(lifecycle.bind(publicThis));
}
```

总结

结束安装过程，主要是初始化我们常见的组件上的选项，只不过我们可以不用 `options` 式的写法，但是实际上源码中仍然是转化成 `options` 处理，主要也是为了兼容 `options` 写法。并且，结束安装的过程比较重要的一点就是调用各个生命周期，而熟悉每个生命周期的执行时机，也可以便于我们平常的开发不犯错。

组件更新过程

> 原文：https://wjchumble.github.io/explain-vue3/chapter4/

基本介绍

基于 Proxy 的响应式原理

基本介绍

> 原文：https://wjchumble.github.io/explain-vue3/chapter5/

> 值得一提的是在 `Vue 3.0` 中没有了`watcher` 的概念，取而代之的是 `effect` ，所以接下来会接触很多和 `effect` 相关的函数

在文章的开始前，我们先准备这样一个简单的 `case`，以便后续分析具体逻辑：

main.js 项目入口

```
import { createApp } from "vue";
import App from "./App.vue";

createApp(App).mount("#app");
```

App.vue 组件

```
<template>
  <button @click="inc">Clicked {{ count }} times.<
</template>

<script>
import { reactive, toRefs } from 'vue'

export default {
  setup() {
    const state = reactive({
      count: 0,
    })
    const inc = () => {
      state.count++
    }
```

```
    return {
      inc,
      ...toRefs(state)
    }
  }
}
</script>
```

安装渲染 Effect

首先，我们大家都知道在通常情况下，我们的页面会使用当前实例的一些属性、计算属性、方法等等。所以，在组件渲染的过程就会发生依赖收集的这个过程。也因此，我们先从组件的渲染过程开始分析。

在组件的渲染过程中，会安装（创建）一个渲染 `effect`，即 `Vue 3.0` 在编译 `template` 的时候，对是否有订阅数据做出相应的判断，创建对应的渲染 `effect`，它的定义如下：

```
const setupRenderEffect = (instance, initialVNode,
    // create reactive effect for rendering
    instance.update = effect(function componentEff
            ....
            instance.isMounted = true;
        }
        else {
            ...
        }
    }, (process.env.NODE_ENV !== 'production') ? c
};
```

我们来大致分析一下 `setupRenderEffect()`。它传入几个参数，它们分别为：

- `instance` 当前 `vm` 实例
- `initialVNode` 可以是组件 `VNode` 或者普通 `VNode`
- `container` 挂载的模板，例如 `div#app` 对应的节点
- `anchor`, `parentSuspense`, `isSVG` 普通情况下都为 `null`

然后在当前实例 `instance` 上创建属性 `update` 赋值为 `effect()` 函数的执行结果，`effect()` 函数传入两个参数：

- `componentEffect()` 函数，它会在具体逻辑之后提到，这里我们先不讲
- `createDevEffectOptions(instance)` 用于后续的派发更新，它会返回一个对象：

```
{
    scheduler: queueJob(job) {
                    if (!queue.includes(job)) {
                        queue.push(job);
                        queueFlush();
                    }
                },
    onTrack: instance.rtc ? e => invokeHooks(insta
    onTrigger: instance.rtg ? e => invokeHooks(ins
}
```

然后，我们再来看看`effect()` 函数定义：

```
function effect(fn, options = EMPTY_OBJ) {
  if (isEffect(fn)) {
    fn = fn.raw;
  }
  const effect = createReactiveEffect(fn, options)
  if (!options.lazy) {
    effect();
  }
```

```
  return effect;
}
```

`effect()` 函数的逻辑较为简单，首先判断是否已经为 `effect`，是则取出之前定义的。不是则通过 `ceateReactiveEffect()` 创建一个 `effect`，而 `creatReactiveEffect()` 的逻辑会是这样：

```
function createReactiveEffect(fn, options) {
  const effect = function reactiveEffect(...args)
    return run(effect, fn, args);
  };
  effect._isEffect = true;
  effect.active = true;
  effect.raw = fn;
  effect.deps = [];
  effect.options = options;
  return effect;
}
```

可以看到在 `createReactiveEffect()` 中先定义了一个 `reactiveEffect()` 函数赋值给 `effect`，它又调用了 `run()` 方法。而 `run()` 方法中传入三个参数，分别为：

- `effect`，即 `reactiveEffect()` 函数本身
- `fn`，即在刚开始 `instance.update` 是调用 `effect` 函数时，传入的函数 `componentEffect()`
- `args` 为一个空数组

并且，对 `effect` 进行了一些初始化，例如我们**最熟悉**的 `Vue 2x` 中的 `deps` 就出现在 `effect` 这个对象上。

然后，我们分析一下 `run()` 函数的逻辑：

```
function run(effect, fn, args) {
  if (!effect.active) {
    return fn(...args);
  }
  if (!effectStack.includes(effect)) {
    cleanup(effect);
    try {
      enableTracking();
      effectStack.push(effect);
      activeEffect = effect;
      return fn(...args);
    } finally {
      effectStack.pop();
      resetTracking();
      activeEffect = effectStack[effectStack.lengt
    }
  }
}
```

在这里，初次创建 `effect`，我们会命中第二个分支逻辑，即当前 `effectStack` 栈中不包含这个 `effect`。那么，首先会执行 `cleanup(effect)`，即遍历`effect.deps`，清空之前的依赖。

> `cleanup()` 的逻辑其实在`Vue 2x`的源码中也有的，避免依赖的重复收集。并且，对比 `Vue 2x`，`Vue 3.0` 中的 `track` 其实相当于 `watcher`，在 `track` 中会进行依赖的收集，后面我们会讲 `track` 的具体实现

然后，执行`enableTracking()`
和`effectStack.push(effect)`，前者的逻辑很简单，即可以追踪，用于后续触发 `track` 的判断：

```
function enableTracking() {
  trackStack.push(shouldTrack);
  shouldTrack = true;
}
```

而后者，即将当前的 `effect` 添加到 `effectStack` 栈中。最后，执行 `fn()` ，即我们一开始定义的 `instance.update = effect()` 时候传入的 `componentEffect()` ：

```
instance.update = effect(function componentEffect(
    if (!instance.isMounted) {
        const subTree = (instance.subTree = render
        // beforeMount hook
        if (instance.bm !== null) {
            invokeHooks(instance.bm);
        }
        if (initialVNode.el && hydrateNode) {
            // vnode has adopted host node - perfo
            hydrateNode(initialVNode.el, subTree,
        }
        else {
            patch(null, subTree, container, anchor
            initialVNode.el = subTree.el;
        }
        // mounted hook
        if (instance.m !== null) {
            queuePostRenderEffect(instance.m, pare
        }
        // activated hook for keep-alive roots.
        if (instance.a !== null &&
            instance.vnode.shapeFlag & 256 /* COMP
            queuePostRenderEffect(instance.a, pare
        }
        instance.isMounted = true;
```

```
    }
    else {
        ...
    }
}, (process.env.NODE_ENV !== 'production') ? creat
```

> 而接下来就会进入组件的渲染过程，其中涉及 `renderComponnetRoot`、`patch` 等等，这次我们并不会分析组件渲染具体细节。

安装渲染 `Effect`，是为后续的依赖收集做一个前期的准备。因为在后面会用到 `setupRenderEffect` 中定义的 `effect()` 函数，以及会调用 `run()` 函数。所以，接下来，我们就正式进入依赖收集部分的分析。

优点

> 原文：https://wjchumble.github.io/explain-vue3/chapter5/reactive

reactive

`reactive` API 的定义为传入一个对象并返回一个基于原对象的响应式代理，即返回一个 `Proxy`，相当于 `Vue2x` 版本中的 `Vue.observer`。

首先，我们需要知道在 `Vue3` 中除了可以使用原先的 `Options API`，还可以使用新的语法 `Composition API`，简易版的 `Composition API` 看起来会是这样的：

```
  setup() {
    const state = reactive({
      count: 0,
      double: computed(() => state.count * 2)
    })

    function increment() {
      state.count++
    }

    return {
      state,
      increment
    }
  }
```

可以看到，没有了我们熟悉的`data`、`computed`、`methods`等等。看起来，似乎有点 `React`风格，这个提出确实当时社区中

引发了很多讨论，说`Vue`越来越像`React`....很多人并不是很能接受，具体细节大家可以去阅读 RFC 的介绍。

回到本篇文章所关注的，很明显 `reactive API` 是**对标** `data` 选项，那么相比较 `data` 选项有哪些优点？

首先，在 `Vue 2x` 中数据的响应式处理是基于 `Object.defineProperty()` 的，但是它只会侦听对象的属性，并不能侦听对象。所以，在添加对象属性的时候，通常需要这样：

```
// vue2x添加属性
Vue.$set(object, "name", wjc);
```

`reactive API` 是基于 `ES2015 Proxy` 实现对**数据对象**的响应式处理，即在 `Vue3` 可以往对象中添加属性，并且这个属性也会具有响应式的效果，例如：

```
// vue3.0中添加属性
object.name = "wjc";
```

注意点

使用 `reactive API` 需要注意的是，当你在 `setup` 中返回的时候，需要通过对象的形式，例如：

```
export default {
  setup() {
    const pos = reactive({
      x: 0,
      y: 0,
    });

    return {
      pos: useMousePosition(),
```

```
    };
  },
};
```

或者，借助 `toRefs API` 包裹一下导出，这种情况下我们就可以使用展开运算符或解构，例如：

```
export default {
  setup() {
    let state = reactive({
      x: 0,
      y: 0,
    });

    state = toRefs(state);
    return {
      ...state,
    };
  },
};
```

> toRefs() 具体做了什么，接下来会和 reactive 一起讲解

源码实现

首先，相信大家都有所耳闻，`Vue3` 用 `TypeScript` 重构了。所以，大家可能会以为这次会看到一堆 `TypeScript` 的类型之类的。出于各种考虑，本次我只是讲解编译后，转为 JS 的源码实现（没啥子门槛，大家放心 hh）。

reactive

1.先来看看 `reactive` 函数的实现：

```
function reactive(target) {
  // if trying to observe a readonly proxy, return
```

```
  if (readonlyToRaw.has(target)) {
    return target;
  }
  // target is explicitly marked as readonly by us
  if (readonlyValues.has(target)) {
    return readonly(target);
  }
  if (isRef(target)) {
    return target;
  }
  return createReactiveObject(
    target,
    rawToReactive,
    reactiveToRaw,
    mutableHandlers,
    mutableCollectionHandlers
  );
}
```

可以，看到先有 3 个逻辑判断，对 `readonly`、`readonlyValues`、`isRef` 分别进行了判断。我们先不看这些逻辑，通常我们定义 `reactive` 会直接传入一个对象。所以会命中最后的逻辑 `createReactiveObject()`。

2.那我们转到 `createReactiveObject()` 的定义：

```
function createReactiveObject(
  target,
  toProxy,
  toRaw,
  baseHandlers,
  collectionHandlers
) {
  if (!isObject(target)) {
    if (process.env.NODE_ENV !== "production") {
```

```
      console.warn(`value cannot be made reactive:
    }
    return target;
  }
  // target already has corresponding Proxy
  let observed = toProxy.get(target);
  if (observed !== void 0) {
    return observed;
  }
  // target is already a Proxy
  if (toRaw.has(target)) {
    return target;
  }
  // only a whitelist of value types can be observ
  if (!canObserve(target)) {
    return target;
  }
  const handlers = collectionTypes.has(target.cons
    ? collectionHandlers
    : baseHandlers;
  observed = new Proxy(target, handlers);
  toProxy.set(target, observed);
  toRaw.set(observed, target);
  return observed;
}
```

`createReactiveObject()` 传入了四个参数，它们分别扮演的角色：

- `target` 是我们定义 `reactive` 时传入的对象
- `toProxy` 是一个空的 `WeakSet`。
- `toProxy` 是一个空的 `WeakSet`。
- `baseHandlers` 是一个已经定义好 `get` 和 `set` 的对象，它看起来会是这样：

```
    const baseHandlers = {
        get(target, key, receiver) {},
        set(target, key, value, receiver) {},
        deleteProxy: (target, key) {},
        has: (target, key) {},
        ownKey: (target) {}
    };
```

- `collectionHandlers` 是一个只包含 get 的对象。

然后，进入 `createReactiveObject()`， 同样地，一些分支逻辑我们这次不会去分析。

> 看源码时需要保持的一个平常心，先看主逻辑

所以，我们会命中最后的逻辑，即：

```
const handlers = collectionTypes.has(target.constr
  ? collectionHandlers
  : baseHandlers;
observed = new Proxy(target, handlers);
toProxy.set(target, observed);
toRaw.set(observed, target);
```

它首先判断 `collectionTypes` 中是否会包含我们传入的 `target` 的构造函数，而 `collectionTypes` 是一个 `Set` 集合，主要包含 `Set`, `Map`, `WeakMap`, `WeakSet` 等四种集合的构造函数。

如果 `collectionTypes` 包含它的构造函数，那么将 `handlers` 赋值为只有 `get` 的 `collectionHandlers` 对象，否则，赋值为 `baseHandlers` 对象。

> 这两者的区别就在于前者只有 get，很显然这个是留给不需要派发更新的变量定义的，例如我们熟悉的 props 它就

> 只实现了 get。

然后，将 `target` 和 `handlers` 传入 `Proxy`，作为参数实例化一个 `Proxy` 对象。这也是我们看到一些文章常谈的 `Vue3` 用 `ES2015 Proxy` 取代了 `Object.defineProperty`。

最后的两个逻辑，也是非常重要，`toProxy()` 将已经定义好 `Proxy` 对象的 `target` 和 对应的 `observed` 作为键值对塞进 `toProxy` 这个 `WeakMap` 中，用于下次如果存在相同引用的 target 需要 reactive，会命中前面的分支逻辑，返回定义之前定义好的 `observed`，即：

```
// target already has corresponding Proxy target 是
let observed = toProxy.get(target);
if (observed !== void 0) {
  return observed;
}
```

而 `toRaw()` 则是和 `toProxy` 相反的键值对存入，用于下次如果传进的 `target` 已经是一个 `Proxy` 对象时，返回这个 `target`，即：

```
// target is already a Proxy target 已经是一个 Proxy
if (toRaw.has(target)) {
  return target;
}
```

toRefs

前面讲了使用 `reactive` 需要关注的点，提及 `toRefs` 可以让我们方便地使用解构和展开运算符，其实是最近 `Vue3 issue` 也有同学讲解过这方面的东西。有兴趣的同学可以移步 When

it's really needed to use `toRefs` in order to retain reactivity of `reactive` value了解。

我当时也凑了一下热闹，如下图：

可以看到，`toRefs` 是在原有 `Proxy` 对象的基础上，返回了一个普通的带有 `get` 和 `set` 的对象。这样就解决了 `Proxy` 对象遇到解构和展开运算符后，失去引用的情况的问题。

总语

好了，对于 `reactive API` 的定义和大致的源码实现就如上面文章中描述的。而分支的逻辑，大家可以自行走不同的 `case` 去阅读。当然，需要说的是这次的源码只是**尝鲜版的**，不排除之后正式的会做诸多优化，但是主体肯定是保持不变的。

get

> 原文：https://wjchumble.github.io/explain-vue3/chapter5/depCollection

依赖收集

> TODO: 待完善

前面，我们已经讲到了在组件渲染过程会安装渲染 `Effect`。然后，进入渲染组件的阶段，即 `renderComponentRoot()`，而此时会调用 `proxyToUse`，即会触发 `runtimeCompiledRenderProxyHandlers` 的 `get`，即：

```
get(target, key) {
    ...
    else if (renderContext !== EMPTY_OBJ && hasOwn
        accessCache[key] = 1 /* CONTEXT */;
        return renderContext[key];
    }
    ...
}
```

可以看出，此时会命中 `accessCache[key] = 1` 和 `renderContext[key]`。对于**前者**是做一个缓存的作用，**后者**是从当前的渲染上下文中获取 `key` 对应的值（（对于本文这个 `case`，`key` 对应的就是 `count`，它的值为 `0`）。

那么，我想这个时候大家会立即反应，此时会触发这个 `count` 对应 `Proxy` 的 `get`。但是，在我们这个 `case` 中，用了 `toRefs()` 将 `reactive` 包裹导出，所以这个触发 `get` 的过程会分为两个阶段：

> 两个阶段的不同点在于，第一阶段的 `target` 为一个 `object`（即上面所说的`toRefs`的对象结构），而第二阶段的 `target` 为`Proxy`对象 `{count: 0}`。具体细节可以看我 上篇文章

`Proxy` 对象`toRefs()` 后得到对象的结构：

```
{
    value: 0
    _isRef: true
    get: function() {}
    set: ƒunction(newVal) {}
}
```

我们先来看看 `get()` 的逻辑：

```
function createGetter(isReadonly = false, shallow
    return function get(target, key, receiver) {
        ...
        const res = Reflect.get(target, key, recei
        if (isSymbol(key) && builtInSymbols.has(ke
            return res;
        }
        ...
        // ref unwrapping, only for Objects, not f
        if (isRef(res) && !isArray(target)) {
            return res.value;
        }
        track(target, "get" /* GET */, key);
        return isObject(res)
            ? isReadonly
                ? // need to lazy access readonly
                    // circular dependency
                    readonly(res)
                : reactive(res)
```

```
            : res;
    };
}
```

> 第一阶段：触发普通对象的 `get`

由于此时是第一阶段，所以我们会命中 `isRef()` 的逻辑，并返回 `res.value`。此时就会触发 `reactive` 定义的 `Proxy` 对象的 `get`。并且需要**注意**的是 `toRefs()` 只能用于对象，否则我们即时触发了 `get` 也不能获取对应的值（这其实也是看源码的一些好处，深度理解 `API` 的使用）。

track

> 第二阶段：触发 `Proxy` 对象的 `get`

此时属于第二阶段，所以我们会命中 `get` 的最后逻辑：

```
track(target, "get" /* GET */, key);
return isObject(res)
  ? isReadonly
    ? // need to lazy access readonly and reactive
      // circular dependency
      readonly(res)
    : reactive(res)
  : res;
```

可以看到，首先会调用 `track()` 函数，进行**依赖收集**，而 `track()` 函数定义如下：

```
function track(target, type, key) {
  if (!shouldTrack || activeEffect === undefined)
    return;
  }
```

```
  let depsMap = targetMap.get(target);
  if (depsMap === void 0) {
    targetMap.set(target, (depsMap = new Map()));
  }
  let dep = depsMap.get(key);
  if (dep === void 0) {
    depsMap.set(key, (dep = new Set()));
  }
  if (!dep.has(activeEffect)) {
    dep.add(activeEffect);
    activeEffect.deps.push(dep);
    if (process.env.NODE_ENV !== "production" && a
      activeEffect.options.onTrack({
        effect: activeEffect,
        target,
        type,
        key,
      });
    }
  }
}
```

可以看到，第一个分支逻辑不会命中，因为我们在前面分析 `run()` 的时候，就已经定义 `ishouldTrack = true` 和 `activeEffect = effect`。然后，命中 `depsMap === void 0` 逻辑，往 `targetMap` 中添加一个键名为 `{count: 0}` 键值为一个空的 `Map`:

```
if (depsMap === void 0) {
  targetMap.set(target, (depsMap = new Map()));
}
```

> 而此时，我们也可以对比`Vue 2.x`，这个 `{count: 0}` 其实就相当于 `data` 选项（以下统称为 `data`）。所以，这里

> 也可以理解成先对 `data` 初始化一个 `Map`，显然这个 `Map` 中存的就是不同属性对应的 `dep`

然后，对 `count` 属性初始化一个 `Map` 插入到 `data` 选项中，即：

```
let dep = depsMap.get(key);
if (dep === void 0) {
  depsMap.set(key, (dep = new Set()));
}
```

所以，此时的 `dep` 就是 `count` 属性对应的主题对象了。接下来，则判断是否当前 `activeEffect` 存在于 `count` 的主题中，如果不存在则往主题 `dep` 中添加 `activeEffect`，并且将当前主题 `dep` 添加到 `activeEffect` 的 `deps` 数组中。

```
if (!dep.has(activeEffect)) {
  dep.add(activeEffect);
  activeEffect.deps.push(dep);
  // 最后的分支逻辑，我们这次并不会命中
}
```

最后，再回到 `get()`，会返回 `res` 的值，在我们这个 `case` 是 `res` 的值是 `0`。

```
return isObject(res)
  ? isReadonly
    ? // need to lazy access readonly and reactive
      // circular dependency
      readonly(res)
    : reactive(res)
  : res;
```

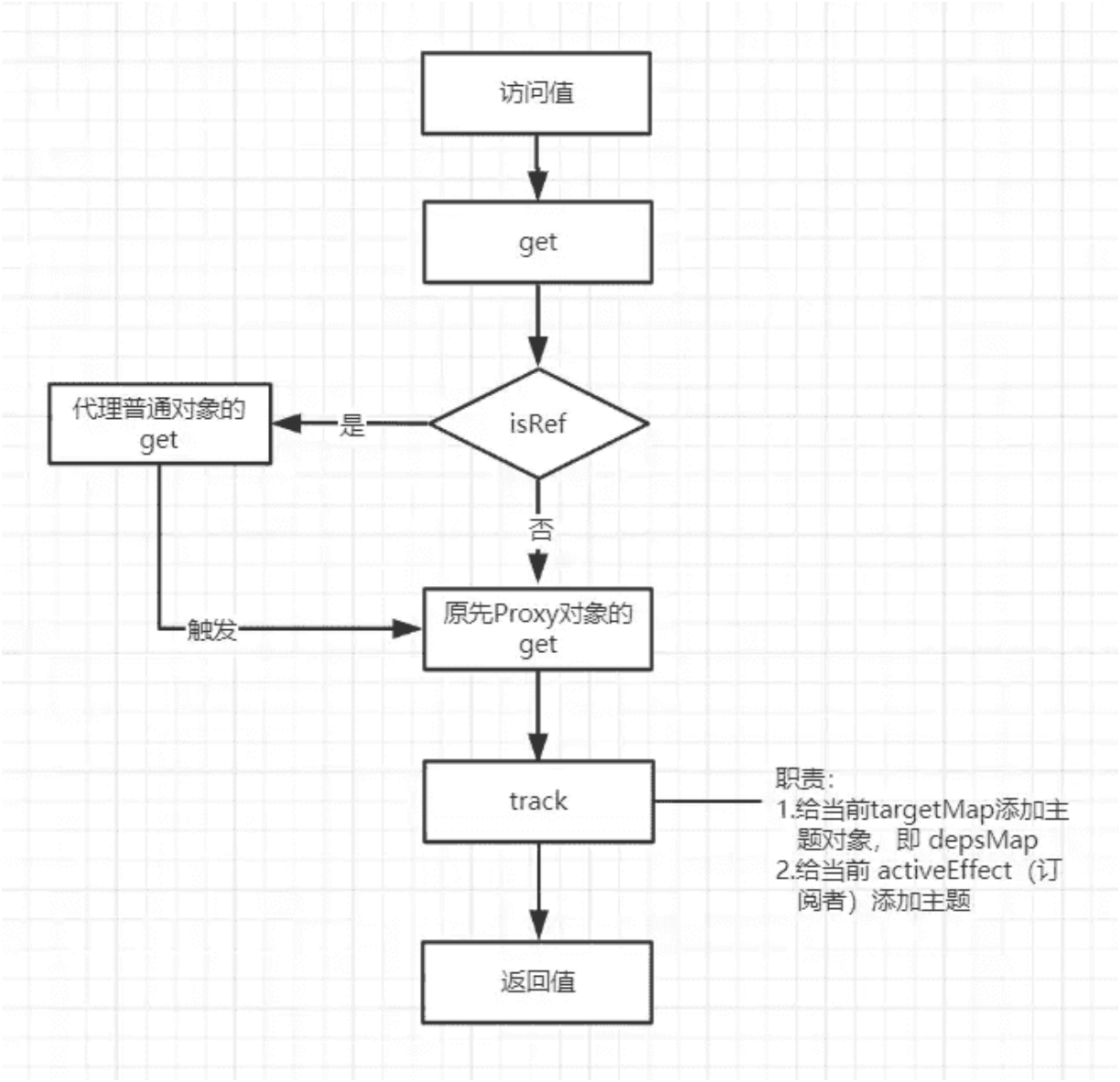
总结

好了，整个 `reactive` 的依赖收集过程，已经分析完了。我们再来回忆其中几个关键点，首先在组件渲染过程，会给当前 `vm` 实例创建一个 `effect`，然后将当前的 `activeEffect` 赋值为 `effect`，并在 `effect` 上创建一些属性，例如非常重要的 `deps` 用于**保存依赖**。

接下来，当该组件使用了 `data` 中的变量时，会访问对应变量的 `get()`。第一次访问 `get()` 会创建 `data` 对应的 `depsMap`，即 `targetMap`。然后再往 `targetMap` 的 `depMap` 中添加对应属性的 `Map`，即 `depsMap`。

创建完属性的 `depsMap` 后，一方面会往该属性的 `depsMap` 中添加当前 `activeEffect`，即**收集订阅者**。另一方面，将该属性的 `depsMap` 添加到 `activeEffect` 的 `deps` 数组中，**即订阅主题**。从而，形成整个依赖收集过程。

> 整个 `get` 过程的流程图


访问值
get
isRef
是
代理普通对象的get
否
触发
原先Proxy对象的get
track
返回值
职责:
1.给当前targetMap添加主题对象，即 depsMap
2.给当前 activeEffect（订阅者）添加主题

set

> 原文：https://wjchumble.github.io/explain-vue3/chapter5/notifyUpdate

trigger

> TODO: 待完善

分析完依赖收集的过程，那么派发更新的整个过程的分析也将会水到渠成。首先，对应派发更新，是指当某个主题发生变化时，在我们这个 `case` 是当 `count` 发生变化时，此时会触发 `data` 的 `set()`，即 `target` 为 `data`，`key` 为 `count`。

```
function set(target, key, value, receiver) {
        ...
        const oldValue = target[key];
        if (!shallow) {
            value = toRaw(value);
            if (!isArray(target) && isRef(oldValue
                oldValue.value = value;
                return true;
            }
        }
        const hadKey = hasOwn(target, key);
        const result = Reflect.set(target, key, va
        // don't trigger if target is something up
        if (target === toRaw(receiver)) {
            if (!hadKey) {
                trigger(target, "add" /* ADD */, k
            }
            else if (hasChanged(value, oldValue))
                trigger(target, "set" /* SET */, k
            }
```

```
        }
        return result;
    };
```

可以看到，`oldValue` 为 `0`，而我们的 `shallow` 此时为 `false`，`value` 为 1。那么，我们看一下 `toRaw()` 函数的逻辑：

```
function toRaw(observed) {
  return reactiveToRaw.get(observed) || readonlyTo
}
```

`toRaw()` 中有两个 `WeakMap` 类型的变量 `reactiveToRaw` 和 `readonlyRaw`。前者是在初始化 `reactive` 的时候，将对应的 `Proxy` 对象存入 `reactiveToRaw` 这个 `Map` 中。后者，则是**存入和前者相反的键值对**。即：

```
function createReactiveObject(target, toProxy, toR
    ...
    observed = new Proxy(target, handlers);
    toProxy.set(target, observed);
    toRaw.set(observed, target);
    ...
}
```

很显然对于 `toRaw()` 方法而言，会返回 `observer` 即 1。所以，**回到 `set()` 的逻辑**，调用 `Reflect.set()` 方法将 `data` 上的 `count` 的值修改为 1。并且，接下来我们还会命中 `target === toRaw(receiver)` 的逻辑。

而 `target === toRaw(receiver)` 的逻辑会处理两个逻辑：

- 如果当前对象不存在该属性，触发 `triger()` 函数对应的 `add`。
- 或者该属性发生变化，触发 `triger()` 函数对应的 `set`

trigger

首先，我们先看一下 `trigger()` 函数的定义：

```
function trigger(target, type, key, newValue, oldV
    const depsMap = targetMap.get(target);
    if (depsMap === void 0) {
        // never been tracked
        return;
    }
    const effects = new Set();
    const computedRunners = new Set();
    if (type === "clear" /* CLEAR */) {
        ...
    }
    else if (key === 'length' && isArray(target))
        ...
    }
    else {
        // schedule runs for SET | ADD | DELETE
        if (key !== void 0) {
            addRunners(effects, computedRunners, d
        }
        // also run for iteration key on ADD | DEL
        if (type === "add" /* ADD */ ||
            (type === "delete" /* DELETE */ && !is
            (type === "set" /* SET */ && target in
            const iterationKey = isArray(target) ?
            addRunners(effects, computedRunners, d
        }
    }
```

```
    const run = (effect) => {
        scheduleRun(effect, target, type, key, (pr
            ? {
                newValue,
                oldValue,
                oldTarget
            }
            : undefined);
    };
    // Important: computed effects must be run fir
    // can be invalidated before any normal effect
    computedRunners.forEach(run);
    effects.forEach(run);
}
```

> 并且，大家可以看到这里有一个细节，就是计算属性的派发更新要优先于普通属性。

在 `trigger()` 函数，首先获取当前 `targetMap` 中 `data` 对应的主题对象的 `depsMap`，而这个 `depsMap` 即我们在依赖收集时在 `track` 中定义的。

然后，初始化两个 `Set` 集合 `effects` 和 `computedRunners`，用于记录普通属性或计算属性的 `effect`，这个过程是会在 `addRunners()` 中进行。

接下来，定义了一个 `run()` 函数，包裹了 `scheduleRun()` 函数，并对开发环境和生产环境进行不同参数的传递，这里由于我们处于开发环境，所以传入的是一个对象，即：

```
{
    newValue: 1,
    oldValue: 0,
    oldTarget: undefined
}
```

然后遍历 `effects`，调用 `run()` 函数，而这个过程实际调用的是 `scheduleRun()`：

```
function scheduleRun(effect, target, type, key, ex
  if (process.env.NODE_ENV !== "production" && eff
    const event = {
      effect,
      target,
      key,
      type,
    };
    effect.options.onTrigger(extraInfo ? extend(ev
  }
  if (effect.options.scheduler !== void 0) {
    effect.options.scheduler(effect);
  } else {
    effect();
  }
}
```

此时，我们会命中 `effect.options.scheduler !== void 0` 的逻辑。然后，调用 `effect.options.scheduler()` 函数，即调用 `queueJob()` 函数：

> `scheduler` 这个属性是在 `setupRenderEffect` 调用 `effect` 函数时创建的。

```
function queueJob(job) {
  if (!queue.includes(job)) {
    queue.push(job);
    queueFlush();
  }
}
```

> 这里使用了一个队列维护所有 `effect()` 函数，其实也和 `Vue 2x` 相似，因为我们 `effect()` 相当于 `watcher`，而 `Vue 2x` 中对 `watcher` 的调用也是通过队列的方式维护。队列的存在具体是为了保持 `watcher` 触发的次序，例如先父 `watcher` 后子 `watcher`。

可以看到 我们会先将 `effect()` 函数添加到队列 `queue` 中，然后调用 `queueFlush()` 清空和调用 `queue`：

```
function queueFlush() {
  if (!isFlushing && !isFlushPending) {
    isFlushPending = true;
    nextTick(flushJobs);
  }
}
```

熟悉 `Vue 2x` 源码的同学，应该知道 `Vue 2x` 中的 `watcher` 也是在下一个 `tick` 中执行，而 `Vue 3.0` 也是一样。而 `flushJobs` 中就会对 `queue` 队列中的 `effect()` 进行执行：

```
function flushJobs(seen) {
  isFlushPending = false;
  isFlushing = true;
  let job;
  if (process.env.NODE_ENV !== "production") {
    seen = seen || new Map();
  }
  while ((job = queue.shift()) !== undefined) {
    if (job === null) {
      continue;
    }
    if (process.env.NODE_ENV !== "production") {
      checkRecursiveUpdates(seen, job);
    }
```

```
    callWithErrorHandling(job, null, 12 /* SCHEDUL
  }
  flushPostFlushCbs(seen);
  isFlushing = false;
  if (queue.length || postFlushCbs.length) {
    flushJobs(seen);
  }
}
```

`flushJob()` 主要会做几件事：

- 首先初始化一个 `Map` 集合 `seen`，然后在递归 `queue` 队列的过程，调用 `checkRecursiveUpdates()` 记录该 `job` 即 `effect()` 触发的次数。如果超过 `100` 次会抛出错误。
- 然后调用 `callWithErrorHandling()`，执行 `job` 即 `effect()`，而我们都知道的是这个 `effect` 是在 `createReactiveEffect()` 时创建的 `reactiveEffect()`，所以，最终会执行 `run()` 方法，即执行最初在 `setupRenderEffectect` 定义的 `effect()`：

```
    const setupRenderEffectect = (instance, initia
        // create reactive effect for rendering
        instance.update = effect(function componen
            if (!instance.isMounted) {
                ...
            }
            else {
                ...
                const nextTree = renderComponentRo
                const prevTree = instance.subTree;
                instance.subTree = nextTree;
                if (instance.bu !== null) {
                    invokeHooks(instance.bu);
                }
```

```
                if (instance.refs !== EMPTY_OBJ) {
                    instance.refs = {};
                }
                patch(prevTree, nextTree,
                hostParentNode(prevTree.el),
                getNextHostNode(prevTree), instanc
                instance.vnode.el = nextTree.el;
                if (next === null) {
                    updateHOCHostEl(instance, next
                }
                if (instance.u !== null) {
                    queuePostRenderEffect(instance
                }
                if ((process.env.NODE_ENV !== 'pro
                    popWarningContext();
                }
            }
        }, (process.env.NODE_ENV !== 'production')
    };
```

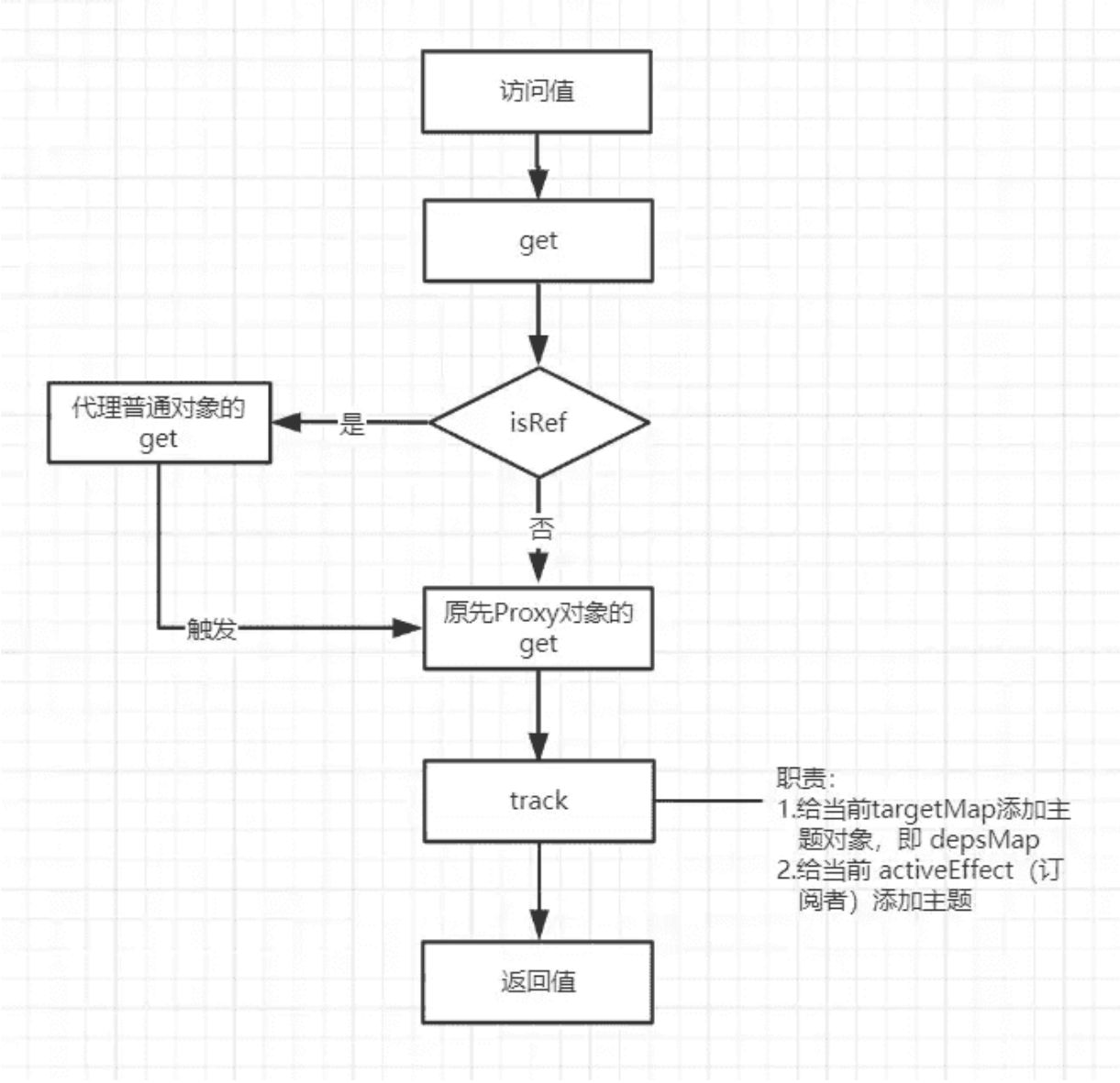
即此时就是派发更新的最后阶段了，会先 `renderComponentRoot()` 创建组件 `VNode`，然后 `patch()` ，即走一遍组件渲染的过程（当然此时称为更新更为贴切）。从而，完成视图的更新。

总结

同样地，我们也来回忆派发更新过程的几个关键点。首先，触发依赖的 `set()`，它会调用 `Reflect.set()` 修改依赖对应属性的值。然后，调用 `trigger()` 函数，获取 `targetMap` 中对应属性的主题，即 `depsMap()`，并且将 `depsMap` 中的 `effect()` 存进 `effect` 集合中。接下来，就将 `effect` 进队，在下一个 `tick` 中清空和执行所有 `effect`。最后，和在

初始化的时候提及的一样，走组件的更新过程，即 `renderComponent()`、`patch()` 等等。

> 整个 `set` 过程的流程图



内置组件

基本介绍

> 原文：https://wjchumble.github.io/explain-vue3/chapter6/

compile 编译生成的 render 函数

> 原文：https://wjchumble.github.io/explain-vue3/chapter6/teleport

teleport

首先，我们从使用性的角度思考 `teleport` 组件**能带给我们什么价值**？

最经典的回答就是开发中使用 `Modal` 模态框的场景。通常，我们会在中后台的业务开发中频繁地使用到模态框。可能对于中台还好，它们会搞一些 `low code` 来**减少开发成本**，但这也是一般大公司或者技术较强的公司才能实现的。

而实际情况下，我们传统的后台开发，就是会存在频繁地**手动使用** `Modal` 的情况，它看起来会是这样：

```
<div class="page">
  <div class="header">我希望点击我出现弹窗</div>
  <!--假设此处有 100 行代码-->
  ....
  <Modal>
    <div>
      我是 header 希望出的弹窗
    </div>
  </Modal>
</div>
```

这样的代码，凸显出来的问题，就是**脱离了所见即所得**的理念，即我头部希望出现的弹窗，**由于样式的问题**，我需要将 `Modal` 写在最下面。

而 `teleport` 组件的出现，**首当其冲**的就是解决这个问题，仍然还是上面那个栗子，通过 `teleport` 组件我们可以这么写：

```
<div class="page">
  <div class="header">我希望点击我出现弹窗</div>
  <!--弹窗内容-->
  <teleport to="#modal-header">
    <div>
      我是 header 希望出的弹窗
    </div>
  </teleport>
  <!--假设此处有 100 行代码-->
  ....
  <Modal id="modal-header">
  </Modal>
</div>
```

结合 `teleport` 组件使用 `modal`，一方面，我们的弹窗内容，就可以符合我们的正常的思考逻辑。并且，另一方面，也可以充分地提高 `Modal` 组件的**可复用性**，即页面中一个 `Modal` 负责展示不同内容。

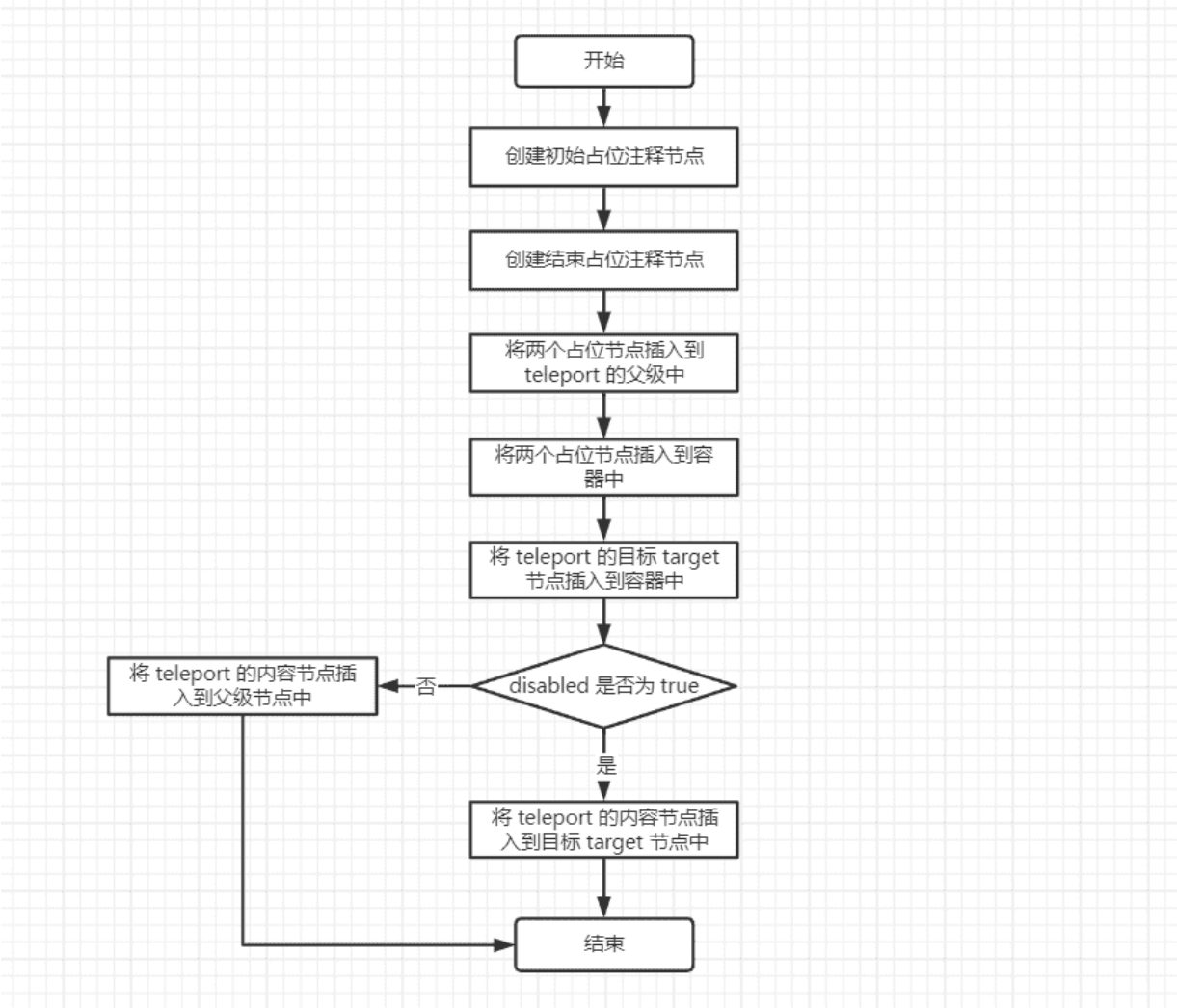
假设，此时我们有一个这样的栗子：

```
<div id="my-heart">
  i love you
</div>
<teleport to="#my-heart" >
  honey
</teleport>
```

通过上面的介绍，我们很容易就知道，它最终渲染到页面上的 DOM 会是这样：

```
<div id="my-heart">i love you honey</div>
```

那么，这个时候我们就会想，`teleport` 组件中的内容，究竟是如何**走进了我的心**？这，说来话长，长话短说，**我们直接上图**：



通过流程图，我们可以知道整体 `teleport` 的工作流并不复杂。那么，接下来，我们再从**源码设计**的角度认识 `teleport` 组件的运行机制。

> 这里，我们仍然会分为 `compile` 和 `runtime` 两个阶段去介绍。

仍然是我们上面的那个栗子，它经过 `compile` 编译处理后生成的**可执行代码**会是这样：

```
const _Vue = Vue
const { createVNode: _createVNode, createTextVNode

const _hoisted_1 = _createVNode("div", { id: "my-h
const _hoisted_2 = _createTextVNode("honey")

return function render(_ctx, _cache) {
  with (_ctx) {
    const { createVNode: _createVNode, createTextV

    return (_openBlock(), _createBlock(_Fragment,
      _hoisted_1,
      (_openBlock(), _createBlock(_Teleport, { to:
        _hoisted_2
      ]))
  ], 64))
}
```

由于，`teleport` 组件并不属于静态节点需要提升的范围，所以它会在 `render` 函数内部创建，即这一部分：

```
_createBlock(_Teleport, { to: "#my-heart" }, [
  _hoisted_2
]))
```

> 需要注意的是，此时 `teleport` 的内容 `honey` 是属于静态节点，所以它会被提升。

并且，这里有一处细节，`teleport` 组件的内部元素永远是**以数组的形式**处理，这在之后的 `patch` 处理中也会提及。

runtime 运行时的 patch 处理

相比较 `compile` 编译时生成 `teleport` 组件的可执行代码，`runtime` 运行时的 `patch` 处理可以说是整个 `teleport` 组件**实现的核心**。

在上一篇文章 深度解读 Vue 3 源码 | compile 和 runtime 结合的 patch 过程 中，我们说了 `patch` 会根据不同的 `shapeFlag` 处理不同的逻辑，而 `teleport` 则会命中 `shapeFlag` 为 `TELEPORT` 的逻辑：

```
function patch(...) {
  ...
  switch(type) {
    ...
    default:
      if (shapeFlag & ShapeFlags.TELEPORT) {
        ;(type as typeof TeleportImpl).process(
          n1 as TeleportVNode,
          n2 as TeleportVNode,
          container,
          anchor,
          parentComponent,
          parentSuspense,
          isSVG,
          optimized,
          internals
        )
      }
  }
}
```

这里会调用 `TeleportImpl` 上的 `process` 方法来实现 `teleport` 的 `patch` 过程，并且它也是 `teleport` 组件实现的**核心代码**。而 `TeleportImpl.process` 函数的逻辑可以分为这四个步骤：

创建并挂载注释节点

首先，创建两个注释 `VNode`，插入此时 `teleport` 组件在页面中的对应位置，即插入到 `teleport` 的父节点 `container` 中：

```
// 创建注释节点
const placeholder = (n2.el = __DEV__
  ? createComment("teleport start")
  : createText(""));
const mainAnchor = (n2.anchor = __DEV__
  ? createComment("teleport end")
  : createText(""));
// 插入注释节点
insert(placeholder, container, anchor);
insert(mainAnchor, container, anchor);
```

挂载 target 节点和占位节点

其次，判断 `teleport` 组件对应 `target` 的 `DOM` 节点是否存在，存在则插入一个**空的文本节点**，也可以称为**占位节点**：

```
const target = (n2.target = resolveTarget(n2.props
const targetAnchor = (n2.targetAnchor = createText
if (target) {
  insert(targetAnchor, target);
} else if (__DEV__) {
  warn("Invalid Teleport target on mount:", target
}
```

定义挂载函数 mount

然后，定义 `mount` 方法来为 `teleport` 组件进行特定的挂载操作，它的本质是基于 `mountChildren` 挂载子元素方法的封装：

```
const mount = (container: RendererElement, anchor:
  if (shapeFlag & ShapeFlags.ARRAY_CHILDREN) {
    mountChildren(
      children as VNodeArrayChildren,
      container,
      anchor,
      parentComponent,
      parentSuspense,
      isSVG,
      optimized
    )
  }
}
```

可以看到，这里也对是否 `ShpeFlags` 为 `ARRAY_CHILDREN`，**即数组**，进行了判断，因为 `teleport` 的**子元素必须为数组**。并且，`mount` 方法的两个形参的意义分别是：

- `container` 代表要挂载的父节点。
- `anchor` 调用 `insertBefore` 插入时的 `referenceNode`，即占位 `VNode`。

根据 disabled 处理不同逻辑

由于，`teleport` 组件提供了一个 `props` 属性 `disabled` 来控制是否将内容显示在目标 `target` 中。所以，最后会根据 `disabled` 来进行不同逻辑的处理：

- `disabled` 为 `true` 时，`mainAnchor` 作为 `referenceNode`，即**注释节点**，挂载到此时 `teleport` 的父级节点中。
- `disabled` 为 `false` 时，`targetAnchor` 作为 `refereneceNode`，即 `target` 中的空文本节点，挂载到此时 `teleport` 的 `target` 节点中。

```
if (disabled) {
  mount(container, mainAnchor);
} else if (target) {
  mount(target, targetAnchor);
}
```

而 `mount` 方法最终会调用原始的 `DOM API` `insertBefore` 来实现 `teleport` 内容的挂载。我们来回忆一下 `insertBefore` 的语法：

```
var insertedNode = parentNode.insertBefore(newNode
```

由于 `insertBefore` 的第二个参数 `referenceNode` 是必选的，**如果不提供节点或者传入无效值，在不同的浏览器中会有不同的表现（摘自 MDN）**。所以，当 `disabled` 为 `false` 时，我们的 `referenceNode` 就是一个已插入 `target` 中的**空文本节点**，从而确保在不同浏览器上都能**表现一致**。

总结

今天介绍的是属于 `teleport` 组件创建的逻辑。同样地，`teleport` 组件也有自己特殊的 `patch` 逻辑，这里有兴趣的同学可以自行去了解。虽说，`teleport` 组件的实现并不复杂，但是，其中的**细节处理仍然是值得学习一番**，例如注释节点来标记 `teleport` 组件位置、空文本节点作为占位节点确保 `insertBefore` 在不同浏览器上表现一致等。

常用指令

基本介绍

> 原文：https://wjchumble.github.io/explain-vue3/chapter7/

派发更新时 patch，更新节点

> 原文：https://wjchumble.github.io/explain-vue3/chapter7/v-if

v-if

在之前模版编译一节中，我给大家介绍了 Vue 3 的编译过程，即一个模版会经历 `baseParse`、`transform`、`generate` 这三个过程，最后由 `generate` 生成可以执行的代码（`render` 函数）。

> 这里，我们就不从编译过程开始讲解 `v-if` 指令的 `render` 函数生成过程了，有兴趣了解这个过程的同学，可以看我之前的模版编译一节

我们可以直接在 Vue3 Template Explore 输入一个使用 `v-if` 指令的栗子：

```
<div v-if="visible"></div>
```

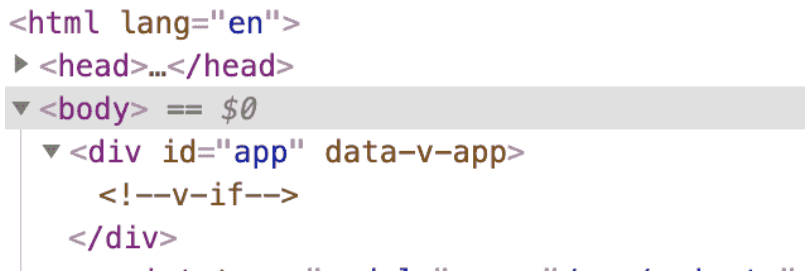
然后，由它编译生成的 `render` 函数会是这样：

```
render(_ctx, _cache, $props, $setup, $data, $optio
  return (_ctx.visible)
    ? (_openBlock(), _createBlock("div", { key: 0
    : _createCommentVNode("v-if", true)
}
```

可以看到，一个简单的使用 `v-if` 指令的模版编译生成的 `render` 函数最终会返回一个**三目运算表达式**。首先，让我们先来认识一下其中几个变量和函数的意义：

- `_ctx` 当前组件实例的上下文，即 `this`
- `_openBlock()` 和 `_createBlock()` 用于构造 `Block Tree` 和 `Block VNode`，它们主要用于靶向更新过程
- `_createCommentVNode()` 创建注释节点的函数，通常用于占位

显然，如果当 `visible` 为 `false` 的时候，会在当前模版中创建一个**注释节点**（也可称为占位节点），反之则创建一个真实节点（即它自己）。例如当 `visible` 为 `false` 时渲染到页面上会是这样：



> 在 Vue 中很多地方都运用了注释节点来作为**占位节点**，其目的是在不展示该元素的时候，标识其**在页面中的位置**，以便在 `patch` 的时候将该元素放回该位置。

那么，这个时候我想大家就会抛出一个疑问：当 `visible` 动态切换 `true` 或 `false` 的这个过程（派发更新）究竟发生了什么？

> 如果不了解 Vue 3 派发更新和依赖收集过程的同学，可以看我之前的文章4k+ 字分析 Vue 3.0 响应式原理（依赖收集和派发更新）

在 Vue 3 中总共有四种指令：`v-on`、`v-model`、`v-show` 和 `v-if`。但是，实际上在源码中，只针对前面三者**进行了特殊处理**，这可以在 `packages/runtime-dom/src/directives` 目录下的文件看出：

```
// packages/runtime-dom/src/directives
|-- driectives
    |-- vModel.ts        ## v-model 指令相关
    |-- vOn.ts           ## v-on 指令相关
    |-- vShow.ts         ## v-show 指令相关
```

而针对 `v-if` 指令是直接走派发更新过程时 `patch` 的逻辑。由于 `v-if` 指令订阅了 `visible` 变量，所以当 `visible` 变化的时候，则会触发**派发更新**，即 `Proxy` 对象的 `set` 逻辑，最后会命中 `componentEffect` 的逻辑。

> 当然，我们也可以称这个过程为组件的更新过程

这里，我们来看一下 `componentEffect` 的定义（伪代码）：

```
// packages/runtime-core/src/renderer.ts
function componentEffect() {
    if (!instance.isMounted) {
        ....
    } else {
        ...
        const nextTree = renderComponentRoot(insta
        const prevTree = instance.subTree
        instance.subTree = nextTree
        patch(
          prevTree,
          nextTree,
          hostParentNode(prevTree.el!)!,
          getNextHostNode(prevTree),
          instance,
          parentSuspense,
          isSVG
        )
        ...
      }
```

```
  }
}
```

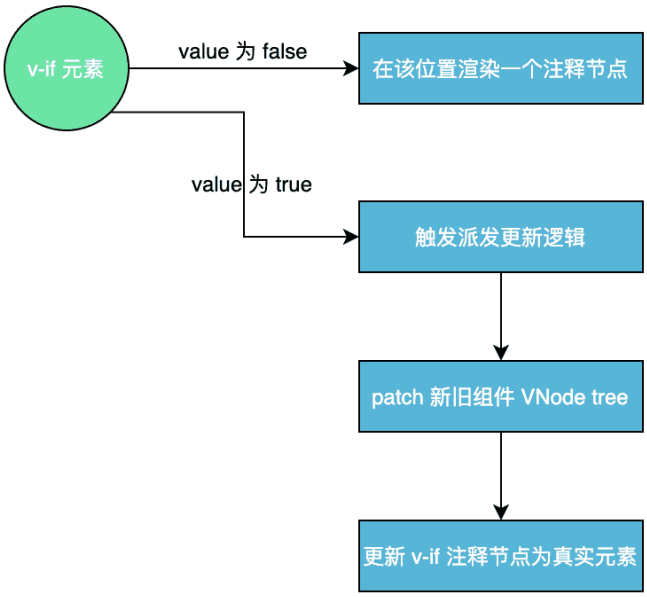
可以看到，当**组件还没挂载时**，即第一次触发派发更新会命中 `!instance.isMounted` 的逻辑。而对于我们这个栗子，则会命中 `else` 的逻辑，即组件更新，主要会做三件事：

- 获取当前组件对应的组件树 `nextTree` 和之前的组件树 `prevTree`
- 更新当前组件实例 `instance` 的组件树 `subTree` 为 `nextTree`
- `patch` 新旧组件树 `prevTree` 和 `nextTree`，如果存在 `dynamicChildren`，即 `Block Tree`，则会命中靶向更新的逻辑，显然我们此时满足条件

> 注：组件树则指的是该组件对应的 VNode Tree。

总结

总体来看，`v-if` 指令的实现较为简单，基于**数据驱动**的理念，当 `v-if` 指令对应的 `value` 为 `false` 的时候会**预先创建一个注释节**点在该位置，然后在 `value` 发生变化时，命中派发更新的逻辑，对新旧组件树进行 `patch`，从而完成使用 `v-if` 指令元素的动态显示隐藏。



> 那么，下一节，我们来看一下 `v-show` 指令的实现 ~

vShow 在生命周期中改变 display 属性

> 原文：https://wjchumble.github.io/explain-vue3/chapter7/v-show

v-show

同样地，对于 `v-show` 指令，我们在 Vue 3 在线模版编译平台输入这样一个栗子：

```
<div v-show="visible"></div>
```

那么，由它编译生成的 `render` 函数：

```
render(_ctx, _cache, $props, $setup, $data, $optio
  return _withDirectives((_openBlock(), _createBlo
  [
    [_vShow, _ctx.visible]
  ])
}
```

此时，这个栗子在 `visible` 为 `false` 时，渲染到页面上的 HTML：

从上面的 `render` 函数可以看出，不同于 `v-if` 的三目运算符表达式，`v-show` 的 `render` 函数返回的是 `_withDirectives()` 函数的执行。

前面，我们已经简单介绍了 `_openBlock()` 和 `_createBlock()` 函数。那么，除开这两者，接下来我们逐点分析一下这个 `render` 函数，首当其冲的是 `vShow` ~

`_vShow` 在源码中则对应着 `vShow`，它被定义在 `packages/runtime-dom/src/directives/vShow`。它的职责是对 `v-show` 指令进行**特殊处理**，主要表现在 `beforeMount`、`mounted`、`updated`、`beforeUnMount` 这四个生命周期中：

```
// packages/runtime-dom/src/directives/vShow.ts
export const vShow: ObjectDirective<VShowElement>
  beforeMount(el, { value }, { transition }) {
    el._vod = el.style.display === 'none' ? '' : e
    if (transition && value) {
      // 处理 tansition 逻辑
      ...
    } else {
      setDisplay(el, value)
    }
  },
  mounted(el, { value }, { transition }) {
    if (transition && value) {
      // 处理 tansition 逻辑
      ...
    }
  },
  updated(el, { value, oldValue }, { transition })
    if (!value === !oldValue) return
    if (transition) {
      // 处理 tansition 逻辑
      ...
    } else {
      setDisplay(el, value)
    }
```

```
  },
  beforeUnmount(el, { value }) {
    setDisplay(el, value)
  }
}
```

对于 `v-show` 指令会处理两个逻辑：普通 `v-show` 或 `transition` 时的 `v-show` 情况。通常情况下我们只是使用 `v-show` 指令，**命中的就是前者**。

> 这里我们只对普通 `v-show` 情况展开分析。

普通 `v-show` 情况，都是调用的 `setDisplay()` 函数，以及会传入两个变量：

- `el` 当前使用 `v-show` 指令的**真实元素**
- `v-show` 指令对应的 `value` 的值

接着，我们来看一下 `setDisplay()` 函数的定义：

```
function setDisplay(el: VShowElement, value: unkno
  el.style.display = value ? el._vod : 'none'
}
```

`setDisplay()` 函数正如它本身**命名的语意**一样，是通过改变该元素的 CSS 属性 `display` 的值来动态的控制 `v-show` 绑定的元素的**显示**或隐藏。

并且，我想大家可能注意到了，当 `value` 为 `true` 的时候，`display` 是等于的 `el.vod`，而 `el.vod` 则等于这个真实元素的 CSS `display` 属性（默认情况下为空）。所以，当 `v-show` 对应的 `value` 为 `true` 的时候，**元素显示与否是取决于它本身**的 CSS `display` 属性。

> 其实，到这里 `v-show` 指令的本质在源码中的体现已经出来了。但是，仍然会留有一些疑问，例如 `withDirectives` 做了什么？`vShow` 在生命周期中对 `v-show` 指令的处理又是如何运用的？

withDirectives 在 VNode 上增加 dir 属性

`withDirectives()` 顾名思义和指令相关，即在 Vue 3 中和指令相关的元素，最后生成的 `render` 函数都会调用 `withDirectives()` 处理指令相关的逻辑，**将 `vShow` 的逻辑作为 `dir` 属性添加**到 `VNode` 上。

`withDirectives()` 函数的定义：

```
// packages/runtime-core/src/directives.ts
export function withDirectives<T extends VNode>(
  vnode: T,
  directives: DirectiveArguments
): T {
  const internalInstance = currentRenderingInstanc
  if (internalInstance === null) {
    __DEV__ && warn(`withDirectives can only be us
    return vnode
  }
  const instance = internalInstance.proxy
  const bindings: DirectiveBinding[] = vnode.dirs
  for (let i = 0; i < directives.length; i++) {
    let [dir, value, arg, modifiers = EMPTY_OBJ] =
    if (isFunction(dir)) {
      ...
    }
    bindings.push({
      dir,
      instance,
      value,
```

```
      oldValue: void 0,
      arg,
      modifiers
    })
  }
  return vnode
}
```

首先，`withDirectives()` 会获取当前渲染实例处理**边缘条件**，即如果在 `render` 函数外面使用 `withDirectives()` 则会抛出异常：

> "withDirectives can only be used inside render functions."

然后，在 `vnode` 上绑定 `dirs` 属性，并且遍历传入的 `directives` 数组，而对于我们这个栗子 `directives` 就是：

```
[
  [_vShow, _ctx.visible]
]
```

显然此时只会**迭代一次**（数组长度为 1）。并且从 `render` 传入的 参数可以知道，从 `directives` 上解构出的 `dir` 指的是 `_vShow`，即我们上面介绍的 `vShow`。由于 `vShow` 是一个对象，所以会重新构造（`bindings.push()`）一个 `dir` 给 `VNode.dir`。

`VNode.dir` 的作用体现在 `vShow` 在生命周期改变元素的 CSS `display` 属性，而这些**生命周期会作为派发更新的结束回调被调用**。

> 接下来，我们一起来看看其中的调用细节 ~

派发更新时 patch，注册 `postRenderEffect` 事件

相信大家应该都知道 Vue 3 提出了 `patchFlag` 的概念，其用来针对不同的场景来执行对应的 `patch` 逻辑。那么，对于上面这个栗子，我们会命中 `patchElement` 的逻辑。

而对于 `v-show` 之类的指令来说，由于 `Vnode.dir` 上绑定了处理元素 CSS `display` 属性的相关逻辑（ `vShow` 定义好的生命周期处理）。所以，此时 `patchElement()` 中会为注册一个 `postRenderEffect` 事件。

```
// packages/runtime-core/src/renderer.ts
const patchElement = (
    n1: VNode,
    n2: VNode,
    parentComponent: ComponentInternalInstance | n
    parentSuspense: SuspenseBoundary | null,
    isSVG: boolean,
    optimized: boolean
  ) => {
    ...
    // 此时 dirs 是存在的
    if ((vnodeHook = newProps.onVnodeUpdated) || d
      // 注册 postRenderEffect 事件
      queuePostRenderEffect(() => {
        vnodeHook && invokeVNodeHook(vnodeHook, pa
        dirs && invokeDirectiveHook(n2, n1, parent
      }, parentSuspense)
    }
    ...
  }
```

这里我们简单分析一下 `queuePostRenderEffect()` 和 `invokeDirectiveHook()` 函数：

- `queuePostRenderEffect()`，`postRenderEffect` 事件注册是通过 `queuePostRenderEffect()` 函数完成的，因为 `effect` 都是维护在一个队列中（为了保持 `effect` 的有序），这里是 `pendingPostFlushCbs`，所以对于 `postRenderEffect` 也是一样的会被**进队**
- `invokeDirectiveHook()`，由于 `vShow` 封装了对元素 CSS `display` 属性的处理，所以 `invokeDirective()` 的本职是调用指令相关的生命周期处理。并且，需要注意的是此时是**更新逻辑**，所以**只会调用 `vShow` 中定义好的 `update` 生命周期**

flushJobs 的结束（finally）调用 `postRenderEffect`

到这里，我们已经围绕 `v-Show` 介绍完了 `vShow`、`withDirectives`、`postRenderEffect` 等概念。但是，万事具备只欠东风，还缺少一个**调用 `postRenderEffect` 事件的时机**，即处理 `pendingPostFlushCbs` 队列的时机。

在 Vue 3 中 `effect` 相当于 Vue 2.x 的 `watch`。虽然变了个命名，但是仍然保持着一样的调用方式，都是调用的 `run()` 函数，然后由 `flushJobs()` 执行 `effect` 队列。而调用 `postRenderEffect` 事件的时机**则是在执行队列的结束**。

`flushJobs()` 函数的定义：

```
// packages/runtime-core/src/scheduler.ts
function flushJobs(seen?: CountMap) {
  isFlushPending = false
  isFlushing = true
```

```
  if (__DEV__) {
    seen = seen || new Map()
  }
  flushPreFlushCbs(seen)
  // 对 effect 进行排序
  queue.sort((a, b) => getId(a!) - getId(b!))
  try {
    for (flushIndex = 0; flushIndex < queue.length
      // 执行渲染 effect
      const job = queue[flushIndex]
      if (job) {
        ...
      }
    }
  } finally {
    ...
    // postRenderEffect 事件的执行时机
    flushPostFlushCbs(seen)
    ...
  }
}
```

在 `flushJobs()` 函数中会执行三种 `effect` 队列，分别是 `preRenderEffect`、`renderEffect`、`postRenderEffect`，它们各自对应 `flushPreFlushCbs()`、`queue`、`flushPostFlushCbs`。

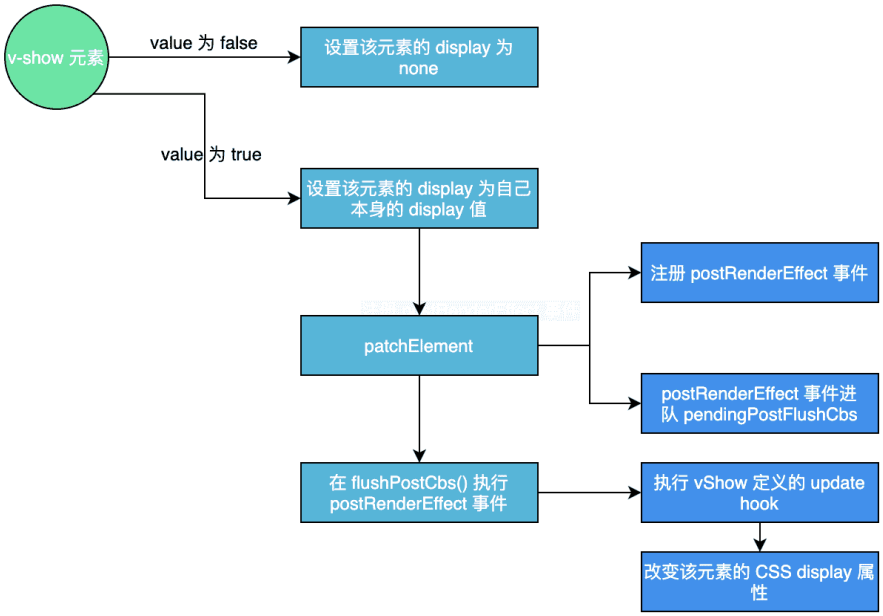
那么，显然 `postRenderEffect` 事件的**调用时机**是在 `flushPostFlushCbs()`。而 `flushPostFlushCbs()` 内部则会遍历 `pendingPostFlushCbs` 队列，即执行之前在 `patchElement` 时注册的 `postRenderEffect` 事件，**本质上就是执行**：

```
updated(el, { value, oldValue }, { transition }) {
  if (!value === !oldValue) return
  if (transition) {
    ...
  } else {
    // 改变元素的 CSS display 属性
    setDisplay(el, value)
  }
},
```

总结

相比较 `v-if` 简单干脆地通过 `patch` 直接更新元素，`v-show` 的处理就略显复杂。这里我们重新梳理一下整个过程：

- 首先，由 `widthDirectives` 来生成最终的 `VNode`。它会给 `VNode` 上绑定 `dir` 属性，即 `vShow` 定义的在生命周期中对元素 CSS `display` 属性的处理
- 其次，在 `patchElement` 的阶段，会注册 `postRenderEffect` 事件，用于调用 `vShow` 定义的 `update` 生命周期处理 CSS `display` 属性的逻辑
- 最后，在派发更新的结束，调用 `postRenderEffect` 事件，即执行 `vShow` 定义的 `update` 生命周期，更改元素的 CSS `display` 属性


v-show 元素
value 为 false
设置该元素的 display 为 none
value 为 true
设置该元素的 display 为自己本身的 display 值
patchElement
注册 postRenderEffect 事件
postRenderEffect 事件进队 pendingPostFlushCbs
在 flushPostCbs() 执行 postRenderEffect 事件
执行 vShow 定义的 update hook
改变该元素的 CSS display 属性

特性

基本介绍

> 原文：https://wjchumble.github.io/explain-vue3/chapter8/

SFC 编译过程处理

> 原文：https://wjchumble.github.io/explain-vue3/chapter8/styleCssVars

Style CSS Variable Injection

Style CSS Variable Injection，即 `<style>` 动态变量注入，根据 SFC 上尤大的总结，它主要有以下 5 点能力：

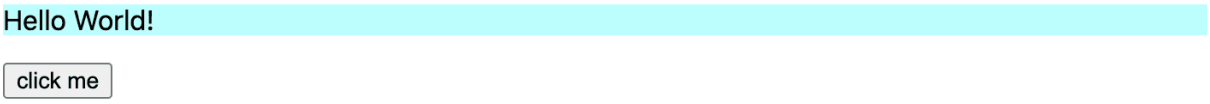
- 不需要明确声明某个属性被注入作为 CSS 变量（会根据 CSS 中的 `v-bind()` 推断）
- 响应式的变量
- 在 Scoped/Non-scoped 模式下具备不同的表现
- 不会污染子组件
- 普通的 CSS 变量的使用不会被影响

下面，我们来看一个简单使用 `<style>` 动态变量注入的例子：

```
<template>
  <p class="word">{{ msg }}</p>
  <button @click="changeColor">click me</button>
</template>

<script setup>
  import { ref } from "vue";

  const msg = "Hello World!";
  let color = ref("red");
  const changeColor = () => {
    if (color.value === "black") {
      color.value = "red";
    } else {
      color.value = "black";
    }
  };
</script>

<style scoped>
  .word {
    background: v-bind(color);
  }
</style>
```

对应的渲染到页面上：

从上面的代码片段，很容易得知当我们点击 `click me` 按钮，文字的背景色就会发生变化：



而这就是 `<style>` 动态变量注入赋予我们的能力，让我们**很便捷地**通过 `<script>` 中的变量来操作 `<template>` 中的 HTML 元素**样式的动态改变**。

那么，这个过程又发生了什么？怎么实现的？有疑问是件好事，接着让我们来一步步揭开其幕后的实现原理。

SFC 在编译过程对 `<style>` 动态变量注入的处理实现，主要是基于的 **2 个关键点**。这里，我们以上面的例子作为示例分析：

- 在对应 DOM 上绑定行内 `style`，通过 `CSS var()` 在 CSS 中使用在行内 `style` 上定义的**自定义属性**，对应的 HTML 部分：

```
<html>
  <head>…</head>
  <body>
    <div id="app" data-v-app>
      <p class="word" data-v-f13b4d11 style="--f13b4d11-color:red;">Hello World!</p> == $0
```

CSS 部分:

```
element.style {
  --f13b4d11-color: red;
}

.word[data-v-f13b4d11] {                                    <style>
  background: var(--f13b4d11-color);
}
```

- 通过**动态更新** `color` 变量来实现行内 `style` 属性值的变化，进而改变使用了该 CSS 自定义属性的 HTML 元素样式

那么，显然要完成这一整个过程，不同于在没有 `<style>` 动态变量注入前的 SFC 编译，这里需要对 `<style>`、`<script>` 增加相应的**特殊处理**。下面，我们分 2 点来讲解：

1.SFC 编译 `<style>` 相关处理

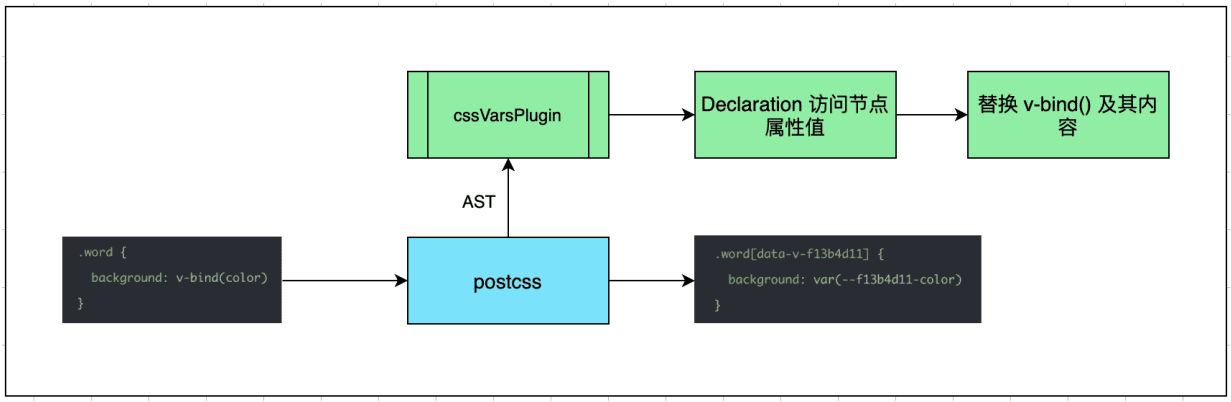
大家都知道的是在 Vue SFC 的 `<style>` 部分编译主要是由 `postcss` 完成的。而这在 Vue 源码中对应着 `packages/compiler-sfc/sfc/compileStyle.ts` 中的 `doCompileStyle()` 方法。

这里，我们看一下其针对 `<style>` 动态变量注入的编译处理，对应的代码（伪代码）：

```
// packages/compiler-sfc/sfc/compileStyle.ts
export function doCompileStyle(
  options: SFCAsyncStyleCompileOptions
): SFCStyleCompileResults | Promise<SFCStyleCompileResults> {
  const {
    ...
    id,
    ...
  } = options
  ...
  const plugins = (postcssPlugins || []).slice()
  plugins.unshift(cssVarsPlugin({ id: shortId, isProd }))
  ...
}
```

可以看到，在使用 `postcss` 编译 `<style>` 之前会加入 `cssVarsPlugin` 插件，并给 `cssVarsPlugin` 传入 `shortId`（即 `scopedId` 替换掉 `data-v` 后的结果）和 `isProd`（是否处于生产环境）。

`cssVarsPlugin` 则是使用了 `postcss` 插件提供的 `Declaration` 方法，来访问 `<style>` 中声明的所有 CSS 属性的值，每次访问通过正则来匹配 `v-bind` 指令的内容，然后再使用 `replace()` 方法将该属性值替换为 `var(--xxxx-xx)`，表现在上面这个例子会是这样：



`cssVarsPlugin` 插件的定义：

```
// packages/compiler-sfc/sfc/cssVars.ts
const cssVarRE = /\bv-bind\(\s*(?:'([^']+)'|"([^"]+)"|([^'"][^)]*))\s
const cssVarsPlugin: PluginCreator<CssVarsPluginOptions> = (opts) =>
```

```
  const { id, isProd } = opts!;
  return {
    postcssPlugin: "vue-sfc-vars",
    Declaration(decl) {
      // rewrite CSS variables
      if (cssVarRE.test(decl.value)) {
        decl.value = decl.value.replace(cssVarRE, (_, $1, $2, $3) =>
          return `var(--${genVarName(id, $1 || $2 || $3, isProd)})`;
        });
      }
    },
  };
};
```

这里 CSS `var()` 的变量名即 `--`（之后的内容）是由 `genVarName()` 方法生成，它会根据 `isProd` 为 `true` 或 `false` 生成不同的值：

```
// packages/compiler-sfc/sfc/cssVars.ts
function genVarName(id: string, raw: string, isProd: boolean): string
  if (isProd) {
    return hash(id + raw);
  } else {
    return `${id}-${raw.replace(/([^\w-])/g, "_")}`;
  }
}
```

2.SFC 编译 `<script>` 相关处理

如果，仅仅站在 `<script>` 的角度，显然是**无法感知**当前 SFC 是否使用了 `<style>` 动态变量注入。所以，需要从 SFC 出发来标识当前是否使用了 `<style>` 动态变量注入。

在 `packages/compiler-sfc/parse.ts` 中的 `parse` 方法中会对解析 SFC 得到的 `descriptor` 对象调用 `parseCssVars()` 方法来获取 `<style>` 中使用到 `v-bind` 的所有变量。

> `descriptor` 指的是解析 SFC 后得到的包含 `script`、`style`、`template` 属性的对象，每个属性包含了 SFC 中每个块（Block）的信息，例如 `<style>` 的属性 `scoped` 和内容等。

对应的 `parse()` 方法中部分代码（伪代码）：

```
// packages/compiler-sfc/parse.ts
function parse(
  source: string,
  {
    sourceMap = true,
```

```
    filename = "anonymous.vue",
    sourceRoot = "",
    pad = false,
    compiler = CompilerDOM,
  }: SFCParseOptions = {}
): SFCParseResult {
  //...
  descriptor.cssVars = parseCssVars(descriptor);
  if (descriptor.cssVars.length) {
    warnExperimental(`v-bind() CSS variable injection`, 231);
  }
  //...
}
```

可以看到，这里会将 `parseCssVars()` 方法返回的结果（数组）赋值给 `descriptor.cssVars`。然后，在编译 `script` 的时候，根据 `descriptor.cssVars.length` 判断是否注入 `<style>` 动态变量注入相关的代码。

而编译 `script` 是由 `package/compile-sfc/src/compileScript.ts` 中的 `compileScript` 方法完成，这里我们看一下其针对 `<style>` 动态变量注入的处理：

```
// package/compile-sfc/src/compileScript.ts
export function compileScript(
  sfc: SFCDescriptor,
  options: SFCScriptCompileOptions
): SFCScriptBlock {
  //...
  const cssVars = sfc.cssVars;
  //...
  const needRewrite = cssVars.length || hasInheritAttrsFlag;
  let content = script.content;
  if (needRewrite) {
    //...
    if (cssVars.length) {
      content += genNormalScriptCssVarsCode(
        cssVars,
        bindings,
        scopeId,
        !!options.isProd
      );
    }
  }
  //...
}
```

对于前面我们举的例子（使用了 `<style>` 动态变量注入），显然 `cssVars.length` 是存在的，所以这里会调用 `genNormalScriptCssVarsCode()` 方法来生成对应的代码。

`genNormalScriptCssVarsCode()` 的定义：

```
// package/compile-sfc/src/cssVars.ts
const CSS_VARS_HELPER = `useCssVars`;
function genNormalScriptCssVarsCode(
  cssVars: string[],
  bindings: BindingMetadata,
  id: string,
  isProd: boolean
): string {
  return (
    `\nimport { ${CSS_VARS_HELPER} as _${CSS_VARS_HELPER} } from 'vue
    `const __injectCSSVars__ = () => {\n${genCssVarsCode(
      cssVars,
      bindings,
      id,
      isProd
    )}}\n` +
    `const __setup__ = __default__.setup\n` +
    `__default__.setup = __setup__\n` +
    `  ? (props, ctx) => { __injectCSSVars__();return __setup__(props
    `  : __injectCSSVars__\n`
  );
}
```

`genNormalScriptCssVarsCode()` 方法主要做了这 3 件事：

- 引入 `useCssVars()` 方法，其主要是监听 `watchEffect` 动态注入的变量，然后再更新对应的 CSS `Vars()` 的值
- 定义 `__injectCSSVars__` 方法，其主要是调用了 `genCssVarsCode()` 方法来生成 `<style>` 动态样式相关的代码
- 兼容非 `<script setup>` 情况下的组合 API 使用（对应这里 `__setup__`），如果它存在则重写 `__default__.setup` 为 `(props, ctx) => { __injectCSSVars__();return __setup__(props, ctx) }`

那么，到这里我们就已经大致分析完 SFC 编译对 `<style>` 动态变量注入的处理，其中部分逻辑并没有过多展开讲解（避免陷入套娃的情况），有兴趣的同学可以自行了解。下面，我们就针对前面这个例子，看一下 SFC 编译结果会是什么？

SFC 编译结果分析

这里，我们直接通过 Vue 官方的 SFC Playground 来查看上面这个例子经过 **SFC 编译**后输出的代码：

```
import { useCssVars as _useCssVars, unref as _unref } from "vue";
import {
  toDisplayString as _toDisplayString,
```

```
  createVNode as _createVNode,
  Fragment as _Fragment,
  openBlock as _openBlock,
  createBlock as _createBlock,
  withScopeId as _withScopeId,
} from "vue";
const _withId = /*#__PURE__*/ _withScopeId("data-v-f13b4d11");

import { ref } from "vue";

const __sfc__ = {
  expose: [],
  setup(__props) {
    _useCssVars((_ctx) => ({
      "f13b4d11-color": _unref(color),
    }));

    const msg = "Hello World!";
    let color = ref("red");
    const changeColor = () => {
      if (color.value === "black") {
        color.value = "red";
      } else {
        color.value = "black";
      }
    };

    return (_ctx, _cache) => {
      return (
        _openBlock(),
        _createBlock(
          _Fragment,
          null,
          [
            _createVNode("p", { class: "word" }, _toDisplayString(msg
            _createVNode("button", { onClick: changeColor }, " click
          ],
          64 /* STABLE_FRAGMENT */
        )
      );
    };
  },
};
__sfc__.__scopeId = "data-v-f13b4d11";
__sfc__.__file = "App.vue";
export default __sfc__;
```

可以看到 SFC 编译的结果，输出了单文件对象 `__sfc__`、`render` 函数、`<style>` 动态变量注入等相关的代码。那么抛开前两者，我们直接看 `<style>` 动态变量注入相关的代码：

```
_useCssVars((_ctx) => ({
  "f13b4d11-color": _unref(color),
}));
```

这里调用了 `_useCssVars()` 方法，即在源码中指的是 `useCssVars()` 方法，然后传入了一个函数，该函数会返回一个对象 `{ "f13b4d11-color": (_unref(color)) }`。那么，下面我们来看一下 `useCssVars()` 方法。

useCssVars

`useCssVars()` 方法是定义在 `runtime-dom/src/helpers/useCssVars.ts` 中：

```
// runtime-dom/src/helpers/useCssVars.ts
function useCssVars(getter: (ctx: any) => Record<string, string>) {
  if (!__BROWSER__ && !__TEST__) return

  const instance = getCurrentInstance()
  if (!instance) {
    __DEV__ &&
      warn(`useCssVars is called without current active component ins
    return
  }

  const setVars = () =>
    setVarsOnVNode(instance.subTree, getter(instance.proxy!))
  onMounted(() => watchEffect(setVars, { flush: 'post' }))
  onUpdated(setVars)
}
```

`useCssVars` 主要做了这 4 件事：

- 获取当前组件实例 `instance`，用于后续操作组件实例的 VNode Tree，即 `instance.subTree`
- 定义 `setVars()` 方法，它会调用 `setVarsOnVNode()` 方法，并将 `instance.subTree`、接收到的 `getter()` 方法传入
- 在 `onMounted()` 生命周期中添加 `watchEffect`，每次挂载组件的时候都会调用 `setVars()` 方法
- 在 `onUpdated()` 生命周期中添加 `setVars()` 方法，每次组件更新的时候都会调用 `setVars()` 方法

可以看到，无论是 `onMounted()` 或者 `onUpdated()` 生命周期，它们都会调用 `setVars()` 方法，本质上也就是 `setVarsOnVNode()` 方法，我们先来看一下它的定

义：

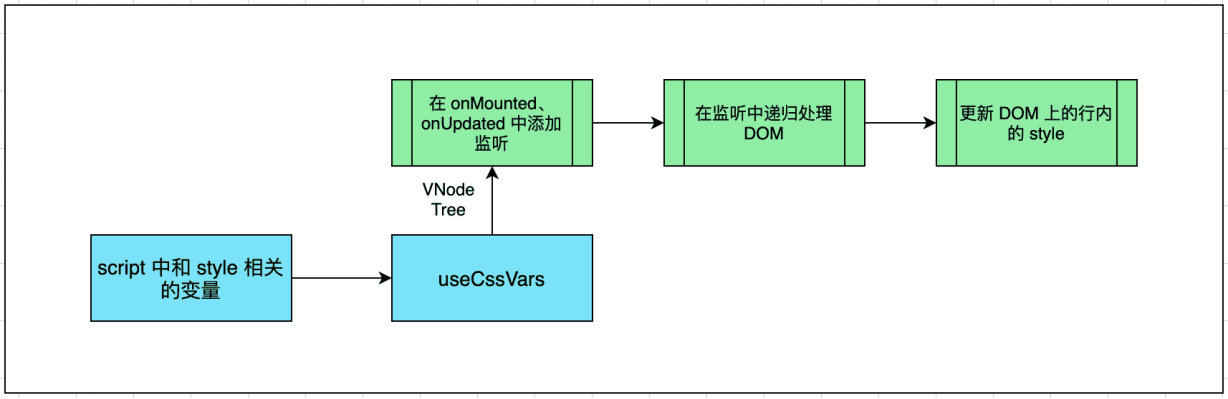
```
// packages/runtime-dom/src/helpers/useCssVars.ts
function setVarsOnVNode(vnode: VNode, vars: Record<string, string>) {
  if (__FEATURE_SUSPENSE__ && vnode.shapeFlag & ShapeFlags.SUSPENSE) {
    const suspense = vnode.suspense!
    vnode = suspense.activeBranch!
    if (suspense.pendingBranch && !suspense.isHydrating) {
      suspense.effects.push(() => {
        setVarsOnVNode(suspense.activeBranch!, vars)
      })
    }
  }

  while (vnode.component) {
    vnode = vnode.component.subTree
  }

  if (vnode.shapeFlag & ShapeFlags.ELEMENT && vnode.el) {
    const style = vnode.el.style
    for (const key in vars) {
      style.setProperty(`--${key}`, vars[key])
    }
  } else if (vnode.type === Fragment) {
    ;(vnode.children as VNode[]).forEach(c => setVarsOnVNode(c, vars))
  }
}
```

对于前面我们这个例子，由于初始传入的是 `instance.subtree`，它的 `type` 为 `Fragment`。所以，在 `setVarsOnVNode()` 方法中会命中 `vnode.type === Fragment` 的逻辑，然后遍历 `vnode.children` 递归调用 `setVarsOnVNode()` 方法。

> 这里不对 `__FEATURE_SUSPENSE__` 和 `vnode.component` 情况做展开分析，有兴趣的同学可以自行了解

而在后续的 `setVarsOnVNode()` 方法的执行，如果满足 `vnode.shapeFlag & ShapeFlags.ELEMENT && vnode.el` 的逻辑，则会调用 `style.setProperty()` 方法来给每个 VNode 对应的 DOM（`vnode.el`）添加行内的 `style`，其中 `key` 是先前处理 `<style>` 时 `CSS var()` 的值，`value` 则对应着 `<script>` 中定义的变量的值。

这样一来，就完成了整个从 `<script>` 中的变量变化到 `<style>` 中样式变化的联动。这里我们用一张图简单回顾一下这个过程：



总结

如果，简单地概括 `<style>` 动态变量注入的话，可能几句话就可以表达。但是，其在源码层面又是怎么做的？这是很值得深入了解的，通过这我们可以懂得如何编写 `postcss` 插件、CSS `vars()` 是什么等技术点。

并且，原本打算留有一个小节用于介绍如何手写一个 Vite 插件 vite-plugin-vue2-css-vars，让 Vue 2.x 也可以支持 `<style>` 动态变量注入。但是，考虑到文章篇幅太长可能会给大家造成阅读上的障碍。所以，这会在下一篇文章中介绍，不过目前这个插件已经发到 NPM 上了，有兴趣的同学也可以自行了解。

语法糖

基本介绍

> 原文：https://wjchumble.github.io/explain-vue3/chapter9/

近期，Vue3 提了一个 Ref Sugar 的 RFC，即 `ref` 语法糖，目前还处理实验性的（Experimental）阶段。在 RFC 的动机（Motivation）中，Evan You 介绍到在 Composition API 引入后，一个主要未解决的问题是 `refs` 和 `reactive` 对象的使用。而到处使用 `.value` 可能会很麻烦，如果在没使用类型系统的情况下，也会很容易错过：

```
let count = ref(1);

function add() {
  count.value++;
}
```

所以，一些用户会更倾向于只使用 `reactive`，这样就不用处理使用 `refs` 的 `.value` 问题。而 `ref` 语法糖的作用是让我们在使用 `ref` 创建响应式的变量时，可以直接获取和更改变量本身，而不是使用 `.value` 来获取和更改对应的值。简单的说，**站在使用层面**，我们可以告别使用 `refs` 时的 `.value` 问题：

```
let count = $ref(1);

function add() {
  count++;
}
```

那么，`ref` 语法糖目前要怎么在项目中使用？它又是怎么实现的？这是我第一眼看到这个 RFC 建立的疑问，相信这也是很多同学持有的疑问。所以，下面让我们来一一揭晓。

Ref 语法糖在项目中的使用

> 原文：https://wjchumble.github.io/explain-vue3/chapter9/ref-sugar

由于 `ref` 语法糖目前还处于实验性的（Experimental）阶段，所以在 Vue3 中不会默认支持 `ref` 语法糖。那么，这里我们以使用 Vite + Vue3 项目开发为例，看一下如何开启对 `ref` 语法糖的支持。

在使用 Vite + Vue3 项目开发时，是由 `@vitejs/plugin-vue` 插件来实现对 `.vue` 文件的代码转换（Transform）、热更新（HMR）等。所以，我们需要在 `vite.config.js` 中给 `@vitejs/plugin-vue` 插件的选项（Options）传入 `refTransform: true`：

```
// vite.config.js
import { defineConfig } from "vite";
import vue from "@vitejs/plugin-vue";

export default defineConfig({
  plugins: [
    vue({
      refTransform: true,
    }),
  ],
});
```

那么，这样一来 `@vitejs/plugin-vue` 插件内部会根据传入的选项中 `refTransform` 的值判断是否需要对 `ref` 语法糖进行特定的代码转换。由于，这里我们设置的是 `true`，显然它是会对 `ref` 语法糖执行特定的代码转换。

接着，我们就可以在 `.vue` 文件中使用 `ref` 语法糖，这里我们看一个简单的例子：

```
<template>
  <div>{{count}}</div>
  <button @click="add">click me</button>
</template>

<script setup>
  let count = $ref(1);

  function add() {
    count++;
  }
</script>
```

对应渲染到页面上：

1
click me

可以看到，我们可以使用 `ref` 语法糖的方式创建响应式的变量，而不用思考使用的时候要加 `.value` 的问题。此外，`ref` 语法糖还支持其他的写法，个人比较推荐的是这里介绍的 `$ref` 的方式，有兴趣的同学可以去 RFC 上了解其他的写法。

那么，在了解完 `ref` 语法糖在项目中的使用后，我们算是解答了第一个疑问（怎么在项目中使用）。下面，我们来解答第二个疑问，它又是怎么实现的，也就是在源码中做了哪些处理？

Ref 语法糖的实现

首先，我们通过 Vue Playground 来直观地感受一下，前面使用 `ref` 语法糖的例子中的 `<script setup>` 块（Block）在编译后的结果：

```
import { ref as _ref } from 'vue'

const __sfc__ = {
  setup(__props) {
  let count = _ref(1)

  function add() {
    count.value++
  }
}
```

可以看到，虽然我们在使用 `ref` 语法糖的时候不需要处理 `.value`，但是它经过编译后**仍然是使用的 `.value`**。那么，这个过程肯定不难免要做很多**编译相关**的代码转换处理。因为，我们需要找到使用 `$ref` 的声明语句和变量，给前者重写为 `_ref`，给后者添加 `.value`。

而在前面，我们也提及 `@vitejs/plugin-vue` 插件会对 `.vue` 文件进行代码的转换，这个过程则是使用的 Vue3 提供的 `@vue/compiler-sfc` 包（Package），它分别提供了对 `<script>`、`<template>`、`<style>` 等块的编译相关的函数。

那么，显然这里我们需要关注的是 `<script>` 块编译相关的函数，这对应的是 `@vue/compiler-sfc` 中的 `compileScript()` 函数。

compileScript() 函数

`compileScript()` 函数定义在 `vue-next` 的 `packages/compiler-sfc/src/compileScript.ts` 文件中，它主要负责对

`<script>` 或 `<script setup>` 块内容的编译处理，它会接收 2 个参数：

- `sfc` 包含 `.vue` 文件的代码被解析后的内容，包含 `script`、`scriptSetup`、`source` 等属性
- `options` 包含一些可选和必须的属性，例如组件对应的 `scopeId` 会作为 `options.id`、前面提及的 `refTransform` 等

`compileScript()` 函数的定义（伪代码）：

```
// packages/compiler-sfc/src/compileScript.ts
export function compileScript(
  sfc: SFCDescriptor,
  options: SFCScriptCompileOptions
): SFCScriptBlock {
  // ...
  return {
    ...script,
    content,
    map,
    bindings,
    scriptAst: scriptAst.body,
  };
}
```

对于 `ref` 语法糖而言，`compileScript()` 函数首先会获取选项（Option）中 `refTransform` 的值，并赋值给 `enableRefTransform`：

```
const enableRefTransform = !!options.refTransform;
```

`enableRefTransform` 则会用于之后判断是否要调用 `ref` 语法糖相关的转换函数。那么，前面我们也提及要使用 `ref` 语法糖，需要先给 `@vite/plugin-vue` 插件选项的 `refTransform` 属性设置为 `true`，它会被传入 `compileScript()` 函数的 `options`，也就是这里的 `options.refTransform`。

接着，会从 `sfc` 中解构出 `scriptSetup`、`source`、`filename` 等属性。其中，会先用源文件的代码字符串 `source` 创建一个 `MagicString` 实例 `s`，它主要会用于后续代码转换时**对源代码字符串进行替换、添加等操作**，然后会调用 `parse()` 函数来解析 `<script setup>` 的内容，即 `scriptSetup.content`，从而生成对应的抽象语法树 `scriptSetupAst`：

```
let { script, scriptSetup, source, filename } = sfc;
const s = new MagicString(source);
const startOffset = scriptSetup.loc.start.offset;
const scriptSetupAst = parse(
  scriptSetup.content,
```

```
  {
    plugins: [...plugins, "topLevelAwait"],
    sourceType: "module",
  },
  startOffset
);
```

而 `parse()` 函数内部则是使用的 `@babel/parser` 提供的 `parser` 方法进行代码的解析并生成对应的 AST。对于上面我们这个例子，生成的 AST 会是这样：

```
{
  body: [ {...}, {...} ],
  directives: [],
  end: 50,
  interpreter: null,
  loc: {
    start: {...},
    end: {...},
    filename: undefined,
    identifierName: undefined
  },
  sourceType: 'module',
  start: 0,
  type: 'Program'
}
```

> 注意，这里省略了 `body`、`start`、`end` 中的内容

然后，会根据前面定义的 `enableRefTransform` 和调用 `shouldTransformRef()` 函数的返回值（`true` 或 `false`）来判断是否进行 `ref` 语法糖的代码转换。如果，需要进行相应的转换，则会调用 `transformRefAST()` 函数来根据 AST 来进行相应的代码转换操作：

```
if (enableRefTransform && shouldTransformRef(scriptSetup.content)) {
  const { rootVars, importedHelpers } = transformRefAST(
    scriptSetupAst,
    s,
    startOffset,
    refBindings
  );
}
```

在前面，我们已经介绍过了 `enableRefTransform`。这里我们来看一下 `shouldTransformRef()` 函数，它主要是通过正则匹配代码内容 `scriptSetup.content` 来判断是否使用了 `ref` 语法糖：

```
// packages/ref-transform/src/refTransform.ts
const transformCheckRE = /[^\w]\$(?:\$|ref|computed|shallowRef)?\(

export function shouldTransform(src: string): boolean {
  return transformCheckRE.test(src);
}
```

所以，当你指定了 `refTransform` 为 `true`，但是你代码中实际并没有使用到 `ref` 语法糖，则在编译 `<script>` 或 `<script setup>` 的过程中也**不会执行**和 `ref` 语法糖相关的代码转换操作，这也是 Vue3 考虑比较细致的地方，避免了不必要的代码转换操作带来性能上的开销。

那么，对于我们这个例子而言（使用了 `ref` 语法糖），则会命中上面的 `transformRefAST()` 函数。而 `transformRefAST()` 函数则对应的是 `packages/ref-transform/src/refTransform.ts` 中的 `transformAST()` 函数。

所以，下面我们来看一下 `transformAST()` 函数是如何根据 AST 来对 `ref` 语法糖相关代码进行转换操作的。

transformAST() 函数

在 `transformAST()` 函数中主要是会遍历传入的原代码对应的 AST，然后通过操作源代码字符串生成的 `MagicString` 实例 `s` 来对源代码进行特定的转换，例如重写 `$ref` 为 `_ref`、添加 `.value` 等。

`transformAST()` 函数的定义（伪代码）：

```
// packages/ref-transform/src/refTransform.ts
export function transformAST(
  ast: Program,
  s: MagicString,
  offset: number = 0,
  knownRootVars?: string[]
): {
  // ...
  walkScope(ast)
  (walk as any)(ast, {
    enter(node: Node, parent?: Node) {
      if (
        node.type === 'Identifier' &&
        isReferencedIdentifier(node, parent!, parentStack) &&
        !excludedIds.has(node)
      ) {
        let i = scopeStack.length
```

```
        while (i--) {
          if (checkRefId(scopeStack[i], node, parent!, parentStack
            return
          }
        }
      }
    }
  })

  return {
    rootVars: Object.keys(rootScope).filter(key => rootScope[key])
    importedHelpers: [...importedHelpers]
  }
}
```

可以看到 `transformAST()` 会先调用 `walkScope()` 来处理根作用域（`root scope`），然后调用 `walk()` 函数逐层地处理 AST 节点，而这里的 `walk()` 函数则是使用的 Rich Haris 写的 `estree-walker`。

下面，我们来分别看一下 `walkScope()` 和 `walk()` 函数做了什么。

walkScope() 函数

首先，这里我们先来看一下前面使用 `ref` 语法糖的声明语句 `let count = $ref(1)` 对应的 AST 结构：

可以看到 `let` 的 AST 节点类型 `type` 会是 `VariableDeclaration`，其余的代码部分对应的 AST 节点则会被放在 `declarations` 中。其中，变量 `count` 的 AST 节点会被作为 `declarations.id` ，而 `$ref(1)` 的 AST 节点会被作为 `declarations.init`。

那么，回到 `walkScope()` 函数，它会根据 AST 节点的类型 `type` 进行特定的处理，对于我们这个例子 `let` 对应的 AST 节点 `type` 为 `VariableDeclaration` 会命中这样的逻辑：

```
function walkScope(node: Program | BlockStatement) {
  for (const stmt of node.body) {
    if (stmt.type === 'VariableDeclaration') {
      for (const decl of stmt.declarations) {
        let toVarCall
        if (
          decl.init &&
          decl.init.type === 'CallExpression' &&
          decl.init.callee.type === 'Identifier' &&
          (toVarCall = isToVarCall(decl.init.callee.name))
        ) {
          processRefDeclaration(
            toVarCall,
            decl.init as CallExpression,
            decl.id,
```

```
            stmt
          )
        }
      }
    }
  }
}
```

这里的 `stmt` 则是 `let` 对应的 AST 节点，然后会遍历 `stmt.declarations`，其中 `decl.init.callee.name` 指的是 `$ref`，接着是调用 `isToVarCall()` 函数并赋值给 `toVarCall`。

`isToVarCall()` 函数的定义：

```
// packages/ref-transform/src/refTransform.ts
const TO_VAR_SYMBOL = "$";
const shorthands = ["ref", "computed", "shallowRef"];
function isToVarCall(callee: string): string | false {
  if (callee === TO_VAR_SYMBOL) {
    return TO_VAR_SYMBOL;
  }
  if (callee[0] === TO_VAR_SYMBOL && shorthands.includes(callee.sl
    return callee;
  }
  return false;
}
```

在前面我们也提及 `ref` 语法糖可以支持其他写法，由于我们使用的是 `$ref` 的方式，所以这里会命中 `callee[0] === TO_VAR_SYMBOL && shorthands.includes(callee.slice(1))` 的逻辑，即 `toVarCall` 会被赋值为 `$ref`。

然后，会调用 `processRefDeclaration()` 函数，它会根据传入的 `decl.init` **提供的位置信息**来对源代码对应的 `MagicString` 实例 `s` 进行操作，即将 `$ref` 重写为 `ref`：

```
// packages/ref-transform/src/refTransform.ts
function processRefDeclaration(
    method: string,
    call: CallExpression,
    id: VariableDeclarator['id'],
    statement: VariableDeclaration
) {
  // ...
  if (id.type === 'Identifier') {
```

```
    registerRefBinding(id)
    s.overwrite(
      call.start! + offset,
      call.start! + method.length + offset,
      helper(method.slice(1))
    )
  }
  // ...
}
```

> 位置信息指的是该 AST 节点在源代码中的位置，通常会用 `start`、`end` 表示，例如这里的 `let count = $ref(1)`，那么 `count` 对应的 AST 节点的 `start` 会是 4、`end` 会是 9。

因为，此时传入的 `id` 对应的是 `count` 的 AST 节点，它会是这样：

```
{
  type: "Identifier",
  start: 4,
  end: 9,
  name: "count"
}
```

所以，这会命中上面的 `id.type === 'Identifier'` 的逻辑。首先，会调用 `registerRefBinding()` 函数，它实际上是调用的是 `registerBinding()`，而 `registerBinding` 会在**当前作用域** `currentScope` 上绑定该变量 `id.name` 并设置为 `true` ，它表示这是一个用 `ref` 语法糖创建的变量，这会用于后续判断是否给某个变量添加 `.value`：

```
const registerRefBinding = (id: Identifier) => registerBinding(id,
function registerBinding(id: Identifier, isRef = false) {
  excludedIds.add(id);
  if (currentScope) {
    currentScope[id.name] = isRef;
  } else {
    error(
      "registerBinding called without active scope, something is w
      id
    );
  }
}
```

可以看到，在 `registerBinding()` 中还会给 `excludedIds` 中添加该 AST 节点，而 `excludeIds` 它是一个 `WeekMap`，它会用于后续跳过不需要进行 `ref` 语法糖处理的类型为 `Identifier` 的 AST 节点。

然后，会调用 `s.overwrite()` 函数来将 `$ref` 重写为 `_ref`，它会接收 3 个参数，分别是重写的起始位置、结束位置以及要重写为的字符串。而 `call` 则对应着 `$ref(1)` 的 AST 节点，它会是这样：

```
{
  type: "Identifier",
  start: 12,
  end: 19,
  callee: {...}
  arguments: {...},
  optional: false
}
```

并且，我想大家应该注意到了在计算重写的起始位置的时候用到了 `offset`，它代表着此时操作的字符串在源字符串中的**偏移位置**，例如该字符串在源字符串中的开始，那么偏移量则会是 `0`。

而 `helper()` 函数则会返回字符串 `_ref`，并且在这个过程会将 `ref` 添加到 `importedHelpers` 中，这会在 `compileScript()` 时用于生成对应的 `import` 语句：

```
function helper(msg: string) {
  importedHelpers.add(msg);
  return `_${msg}`;
}
```

那么，到这里就完成了对 `$ref` 到 `_ref` 的重写，也就是此时我们代码的会是这样：

```
let count = _ref(1);

function add() {
  count++;
}
```

接着，则是通过 `walk()` 函数来将 `count++` 转换成 `count.value++`。下面，我们来看一下 `walk()` 函数。

walk() 函数

前面，我们提及 `walk()` 函数使用的是 Rich Haris 写的 estree-walker，它是一个用于遍历符合 ESTree 规范的 AST 包（Package）。

`walk()` 函数使用起来会是这样：

```
import { walk } from "estree-walker";

walk(ast, {
  enter(node, parent, prop, index) {
    // ...
  },
  leave(node, parent, prop, index) {
    // ...
  },
});
```

可以看到，`walk()` 函数中可以传入 `options`，其中 `enter()` 在每次访问 AST 节点的时候会被调用，`leave()` 则是在离开 AST 节点的时候被调用。

那么，回到前面提到的这个例子，`walk()` 函数主要做了这 2 件事：

1.维护 scopeStack、parentStack 和 currentScope

`scopeStack` 用于存放此时 AST 节点所处的作用域链，初始情况下栈顶为根作用域 `rootScope`；`parentStack` 用于存放遍历 AST 节点过程中的祖先 AST 节点（栈顶的 AST 节点是当前 AST 节点的父亲 AST 节点）；`currentScope` 指向当前的作用域，初始情况下等于根作用域 `rootScope`：

```
const scopeStack: Scope[] = [rootScope];
const parentStack: Node[] = [];
let currentScope: Scope = rootScope;
```

所以，在 `enter()` 的阶段会判断此时 AST 节点类型是否为函数、块，是则**入栈** `scopeStack`：

```
parent && parentStack.push(parent)
if (isFunctionType(node)) {
  scopeStack.push((currentScope = {}))
  // ...
  return
}
if (node.type === 'BlockStatement' && !isFunctionType(parent!)) {
  scopeStack.push((currentScope = {}))
  // ...
  return
}
```

然后，在 `leave()` 的阶段判断此时 AST 节点类型是否为函数、块，是则**出栈** `scopeStack`，并且更新 `currentScope` 为出栈后的 `scopeStack` 的栈顶元素：

```
parent && parentStack.pop()
if (
  (node.type === 'BlockStatement' && !isFunctionType(parent!)) ||
  isFunctionType(node)
) {
  scopeStack.pop()
  currentScope = scopeStack[scopeStack.length - 1] || null
}
```

2.处理 Identifier 类型的 AST 节点

由于，在我们的例子中 `ref` 语法糖创建 `count` 变量的 AST 节点类型是 `Identifier`，所以这会在 `enter()` 阶段命中这样的逻辑：

```
if (
    node.type === 'Identifier' &&
    isReferencedIdentifier(node, parent!, parentStack) &&
    !excludedIds.has(node)
  ) {
    let i = scopeStack.length
    while (i--) {
      if (checkRefId(scopeStack[i], node, parent!, parentStack)) {
        return
      }
    }
  }
```

在 `if` 的判断中，对于 `excludedIds` 我们在前面已经介绍过了，而 `isReferencedIdentifier()` 则是通过 `parenStack` 来判断当前类型为 `Identifier` 的 AST 节点 `node` 是否是一个引用了这之前的某个 AST 节点。

然后，再通过访问 `scopeStack` 来沿着作用域链来判断是否某个作用域中有 `id.name`（变量名 `count`）属性以及属性值为 `true`，这代表它是一个使用 `ref` 语法糖创建的变量，最后则会通过操作 `s`（`s.appendLeft`）来给该变量添加 `.value`：

```
function checkRefId(
    scope: Scope,
    id: Identifier,
    parent: Node,
    parentStack: Node[]
): boolean {
  if (id.name in scope) {
    if (scope[id.name]) {
```

```
      // ...
      s.appendLeft(id.end! + offset, '.value')
    }
    return true
  }
  return false
}
```

总结

通过了解 `ref` 语法糖的实现，我想大家应该会对语法糖这个术语会有不一样的理解，它的本质是在编译阶段通过遍历 AST 来操作特定的代码转换操作。并且，这个实现过程的一些工具包（Package）的配合使用也是非常巧妙的，例如 `MagicString` 操作源代码字符串、`estree-walker` 遍历 AST 节点和作用域相关处理等。

最后，如果文中存在表达不当或错误的地方，欢迎各位同学提 Issue ~