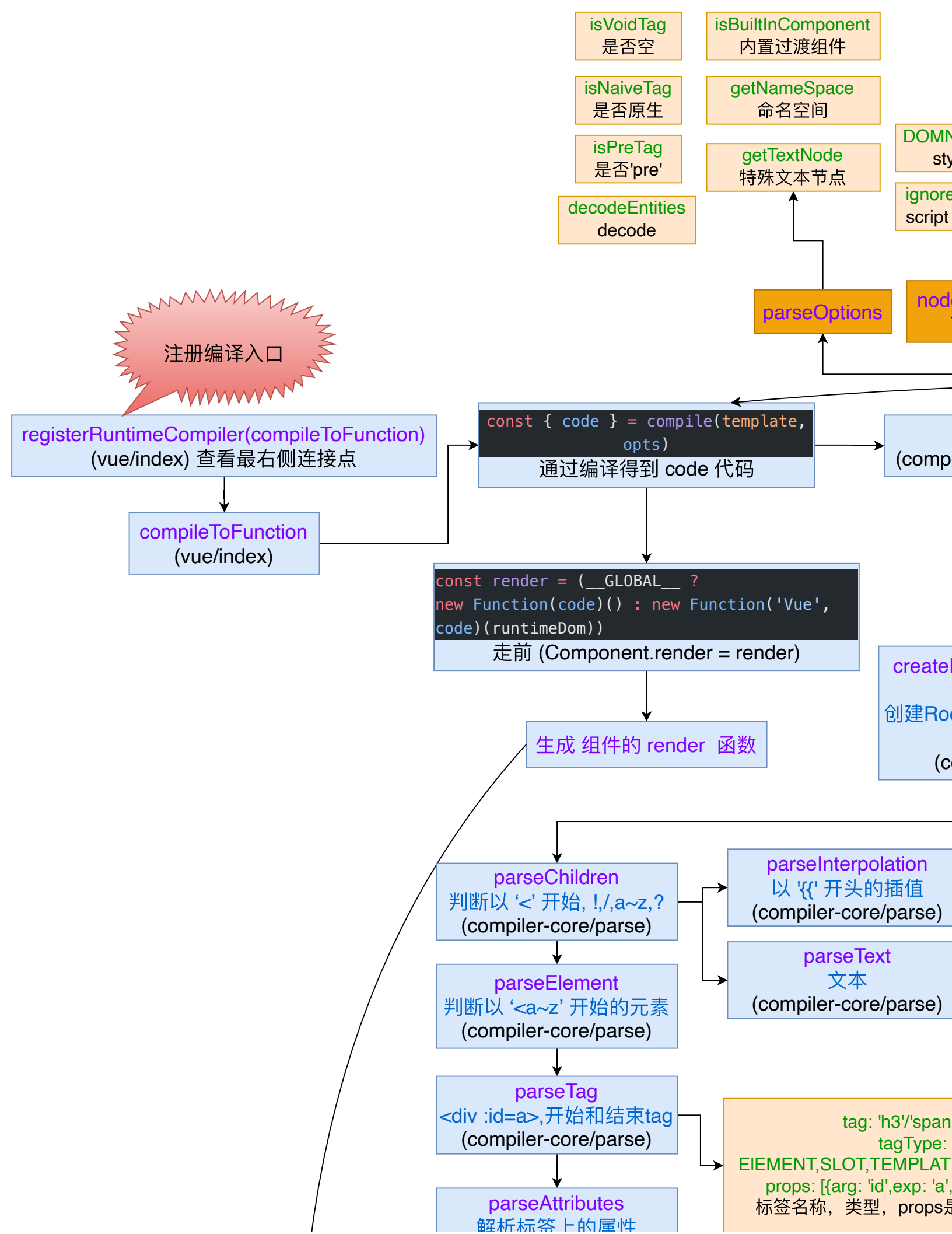
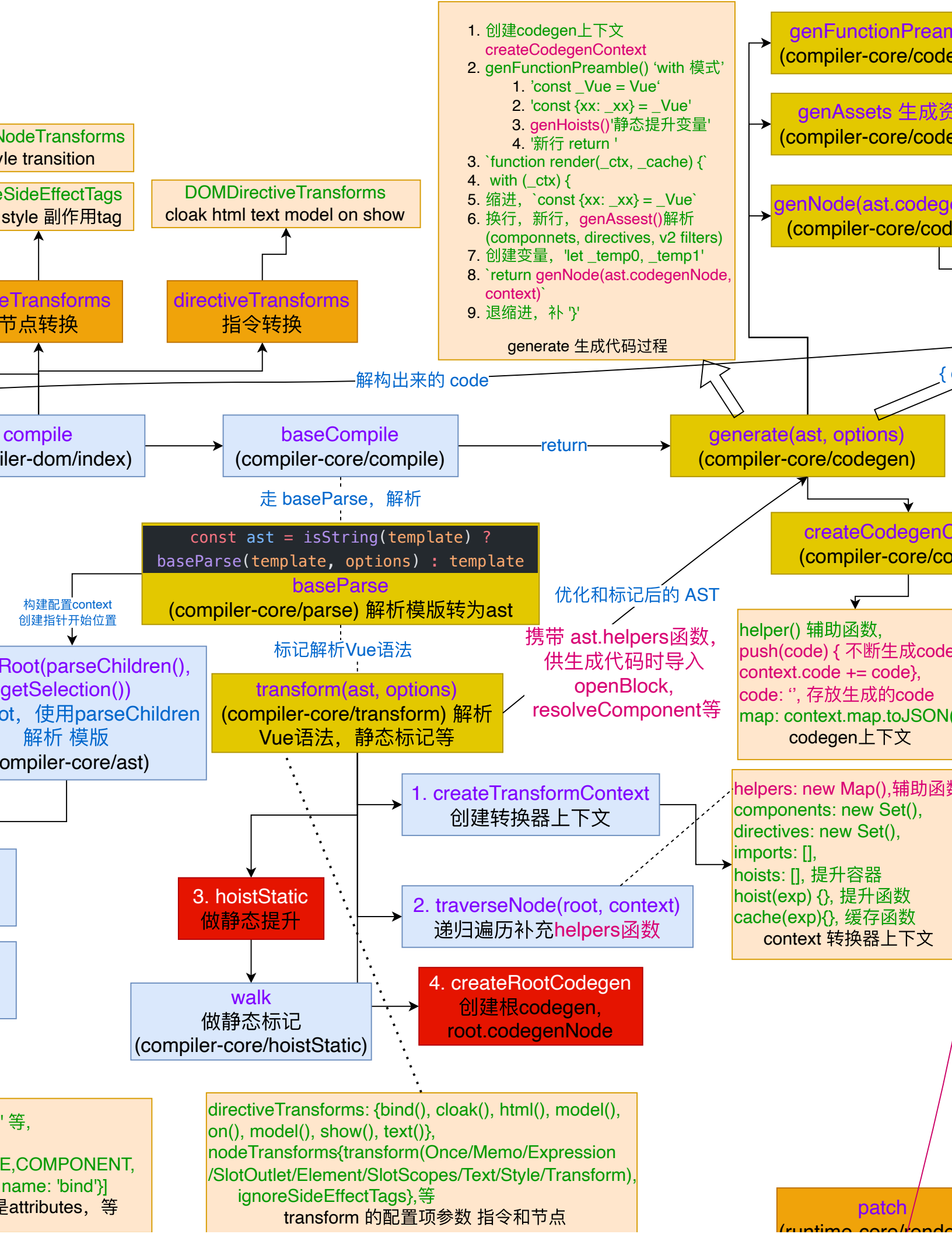
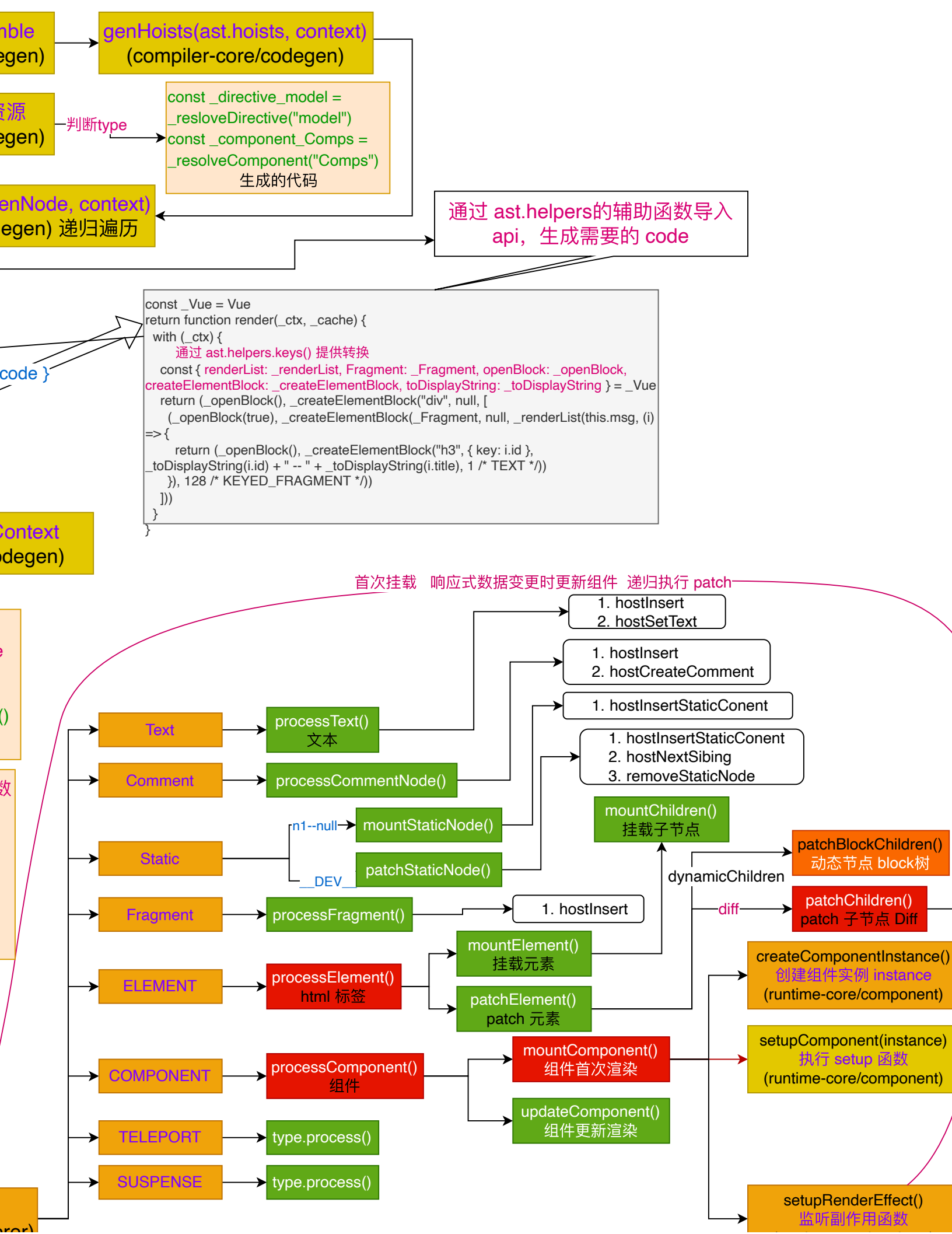
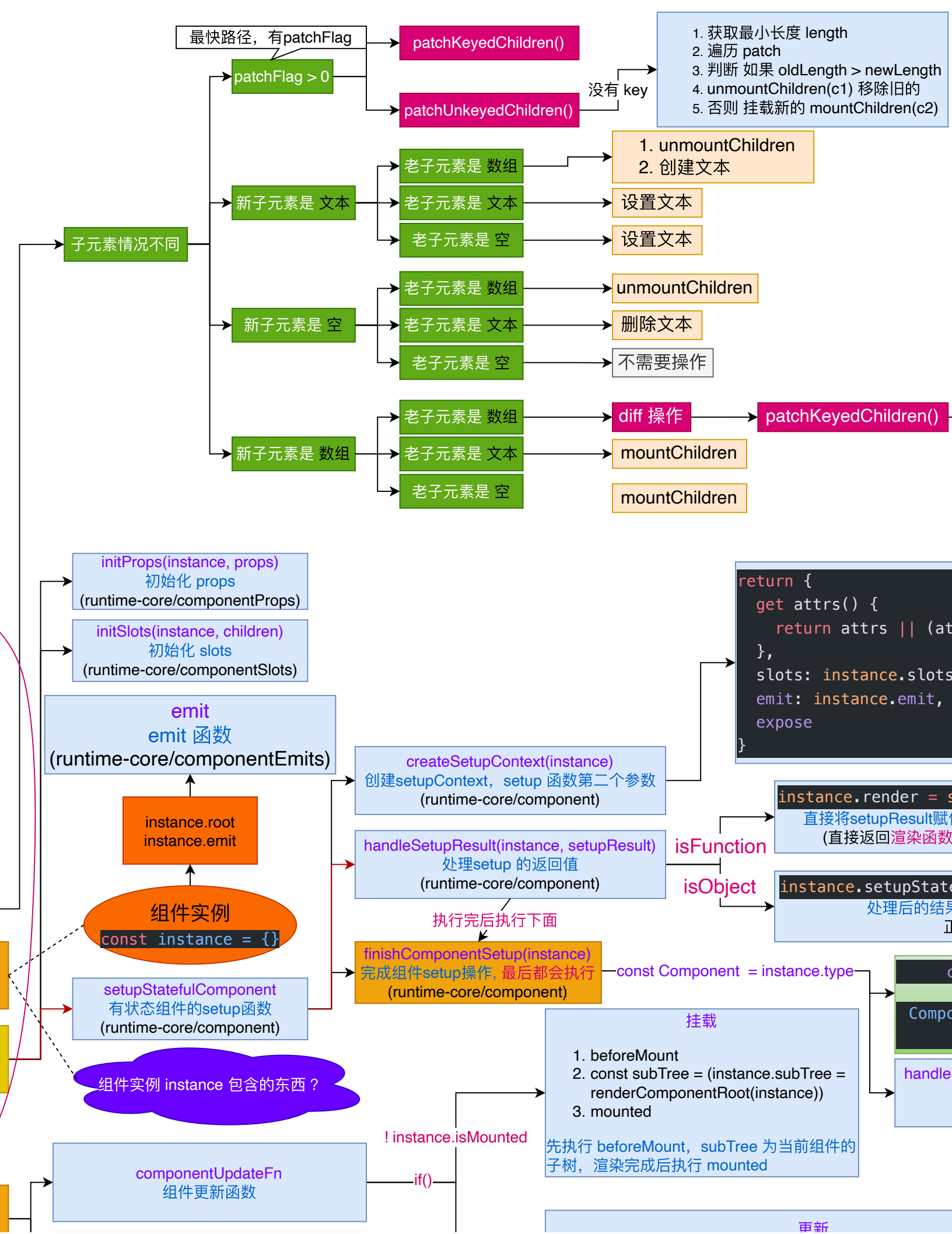
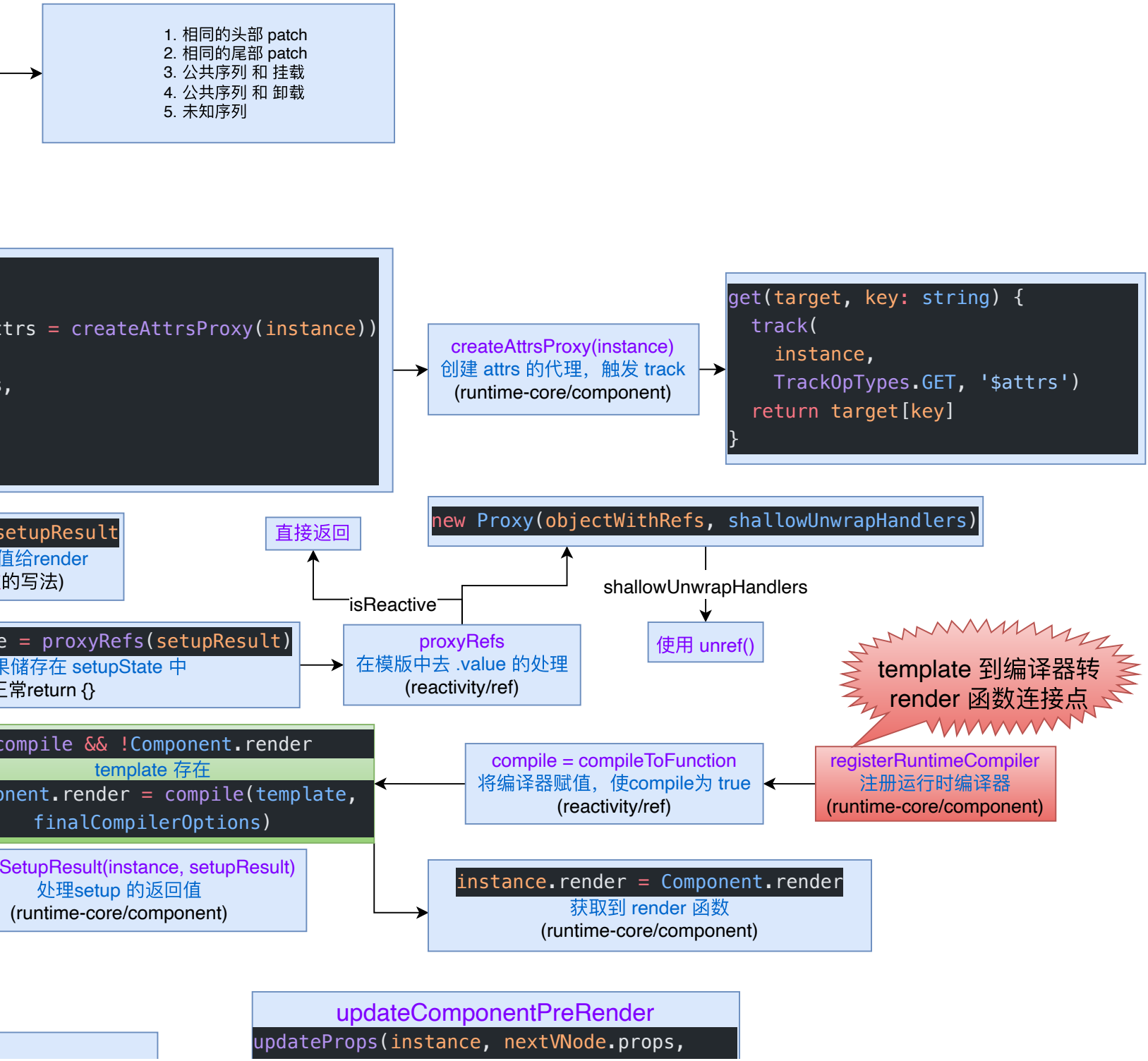
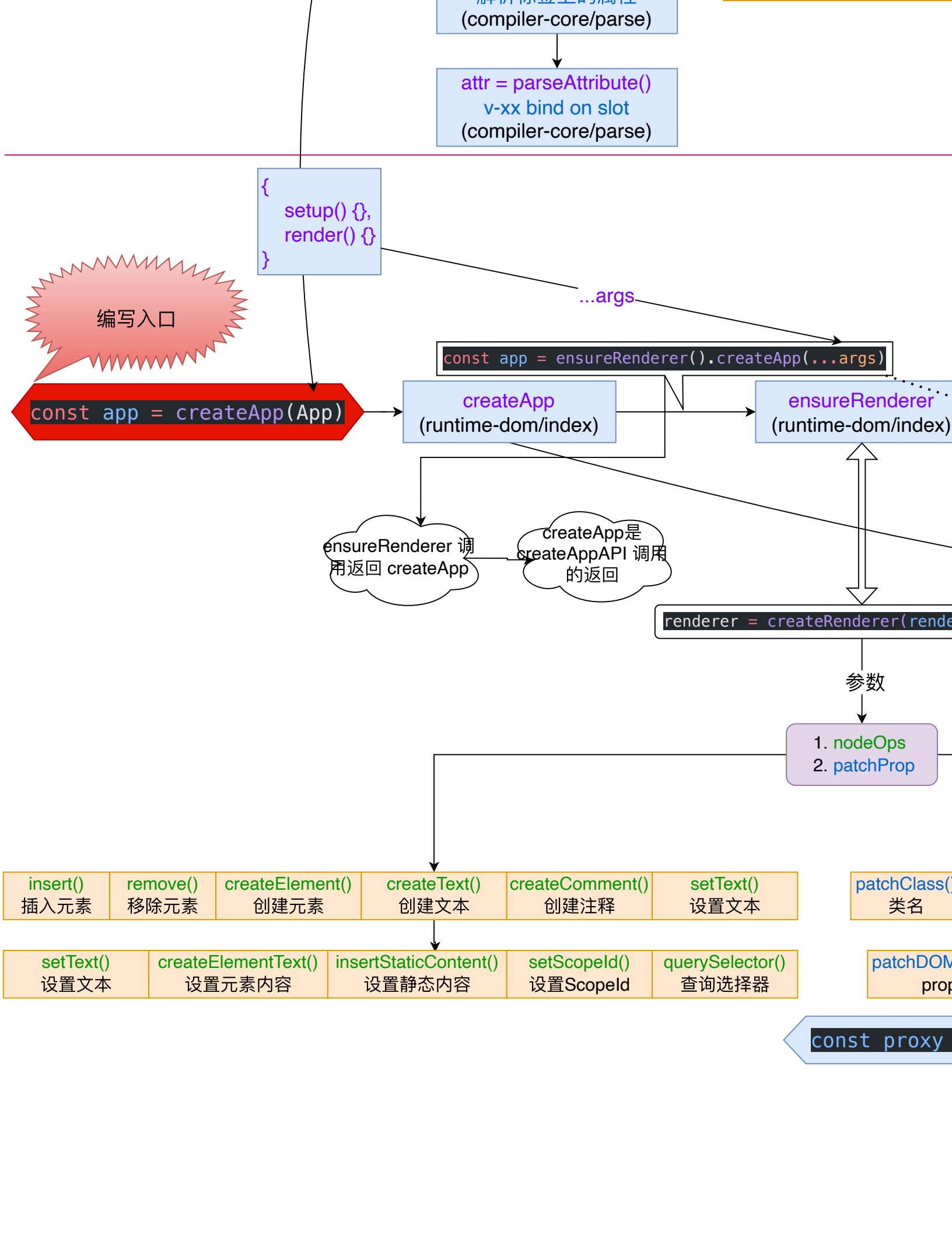

isVoidTag
是否空
isBuiltInComponent
内置过渡组件
isNaiveTag
是否原生
getNameSpace
命名空间
isPreTag
是否'pre'
getTextNode
特殊文本节点
decodeEntities
decode
parseOptions
注册编译入口
registerRuntimeCompiler(compileToFunction)
(vue/index) 查看最右侧连接点
compileToFunction
(vue/index)
const { code } = compile(template, opts)
通过编译得到 code 代码
const render = (__GLOBAL__ ? new Function(code)() : new Function('Vue', code)(runtimeDom))
走前 (Component.render = render)
生成 组件的 render 函数
parseChildren
判断以 '<' 开始, !,/,a~z,?
(compiler-core/parse)
parseInterpolation
以 '{{' 开头的插值
(compiler-core/parse)
parseText
文本
(compiler-core/parse)
parseElement
判断以 '<a~z' 开始的元素
(compiler-core/parse)
parseTag
<div :id=a>,开始和结束tag
(compiler-core/parse)
parseAttributes
解析标签上的属性
tag: 'h3'/'span
tagType:
EIEMENT,SLOT,TEMPLAT
props: [{arg: 'id',exp: 'a',
标签名称，类型，props是

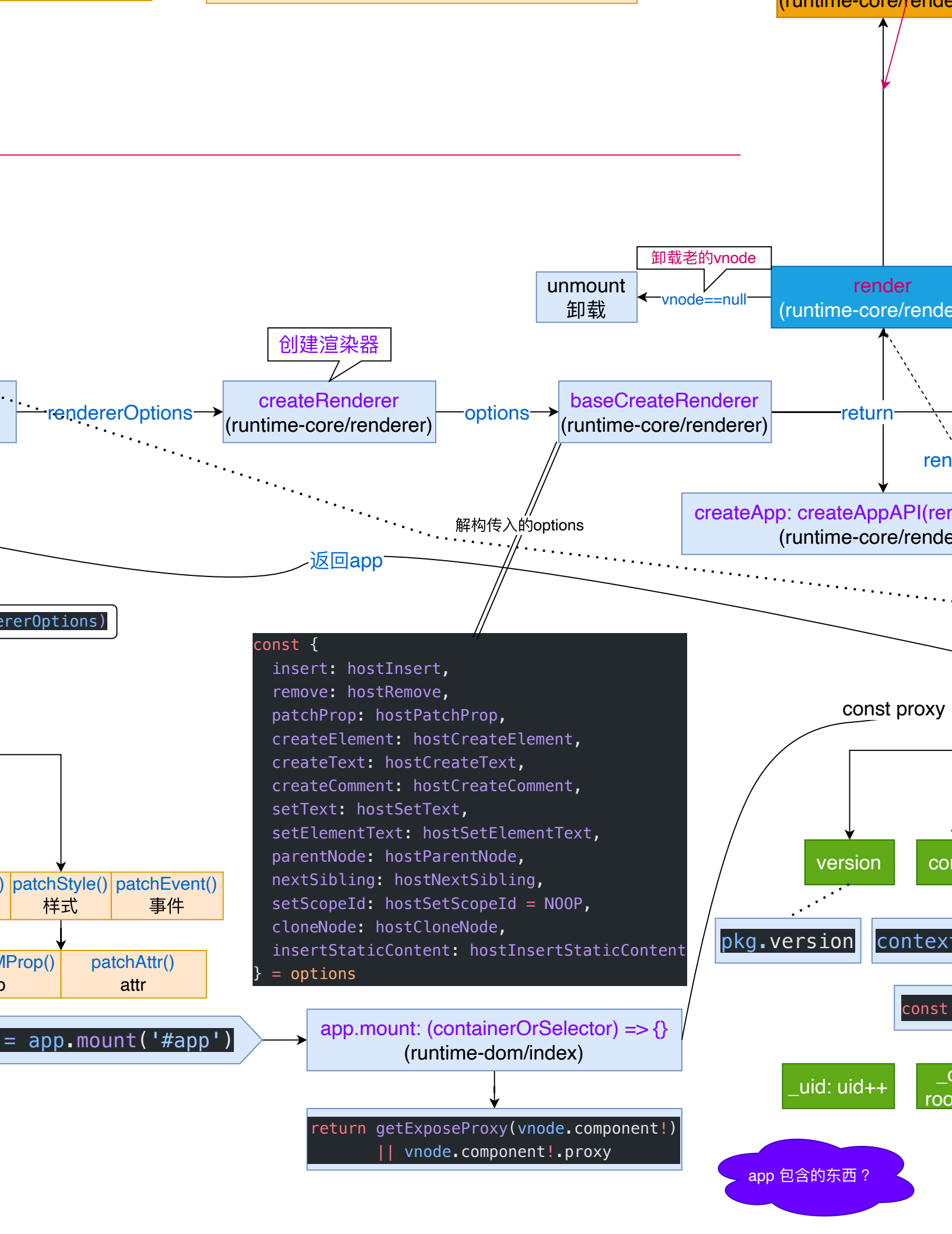

1. 创建codegen上下文 createCodegenContext
2. genFunctionPreamble() 'with 模式'
1. 'const _Vue = Vue'
2. 'const {xx: _xx} = _Vue'
3. genHoists()'静态提升变量'
4. '新行 return '
3. `function render(_ctx, _cache) {`
4. with (_ctx) {
5. 缩进，`const {xx: _xx} = _Vue`
6. 换行，新行，genAssest()解析 (componnets, directives, v2 filters)
7. 创建变量，'let _temp0, _temp1'
8. `return genNode(ast.codegenNode, context)`
9. 退缩进，补 '}'
generate 生成代码过程
genFunctionPrean
(compiler-core/code
genAssets 生成资
(compiler-core/code
genNode(ast.codeg
(compiler-core/cod
NodeTransforms
yle transition
eSideEffectTags
style 副作用tag
DOMDirectiveTransforms
cloak html text model on show
eTransforms
节点转换
directiveTransforms
指令转换
解构出来的 code
compile
iler-dom/index)
baseCompile
(compiler-core/compile)
return
generate(ast, options)
(compiler-core/codegen)
走 baseParse，解析
const ast = isString(template) ? baseParse(template, options) : template
baseParse
(compiler-core/parse) 解析模版转为ast
构建配置context
创建指针开始位置
createCodegenC
(compiler-core/co
优化和标记后的 AST
Root(parseChildren(),
getSelection())
ot，使用parseChildren
解析 模版
ompiler-core/ast)
标记解析Vue语法
transform(ast, options)
(compiler-core/transform) 解析
Vue语法，静态标记等
携带 ast.helpers函数，
供生成代码时导入
openBlock,
resolveComponent等
helper() 辅助函数,
push(code) { 不断生成code
context.code += code},
code: '', 存放生成的code
map: context.map.toJSON(
codegen上下文
1. createTransformContext
创建转换器上下文
helpers: new Map(),辅助函数
components: new Set(),
directives: new Set(),
imports: [],
hoists: [], 提升容器
hoist(exp) {}, 提升函数
cache(exp){}, 缓存函数
context 转换器上下文
3. hoistStatic
做静态提升
2. traverseNode(root, context)
递归遍历补充helpers函数
walk
做静态标记
(compiler-core/hoistStatic)
4. createRootCodegen
创建根codegen,
root.codegenNode
' 等,
E,COMPONENT,
name: 'bind'}]
是attributes，等
directiveTransforms: {bind(), cloak(), html(), model(), on(), model(), show(), text()},
nodeTransforms{transform(Once/Memo/Expression /SlotOutlet/Element/SlotScopes/Text/Style/Transform),
ignoreSideEffectTags},等
transform 的配置项参数 指令和节点
patch

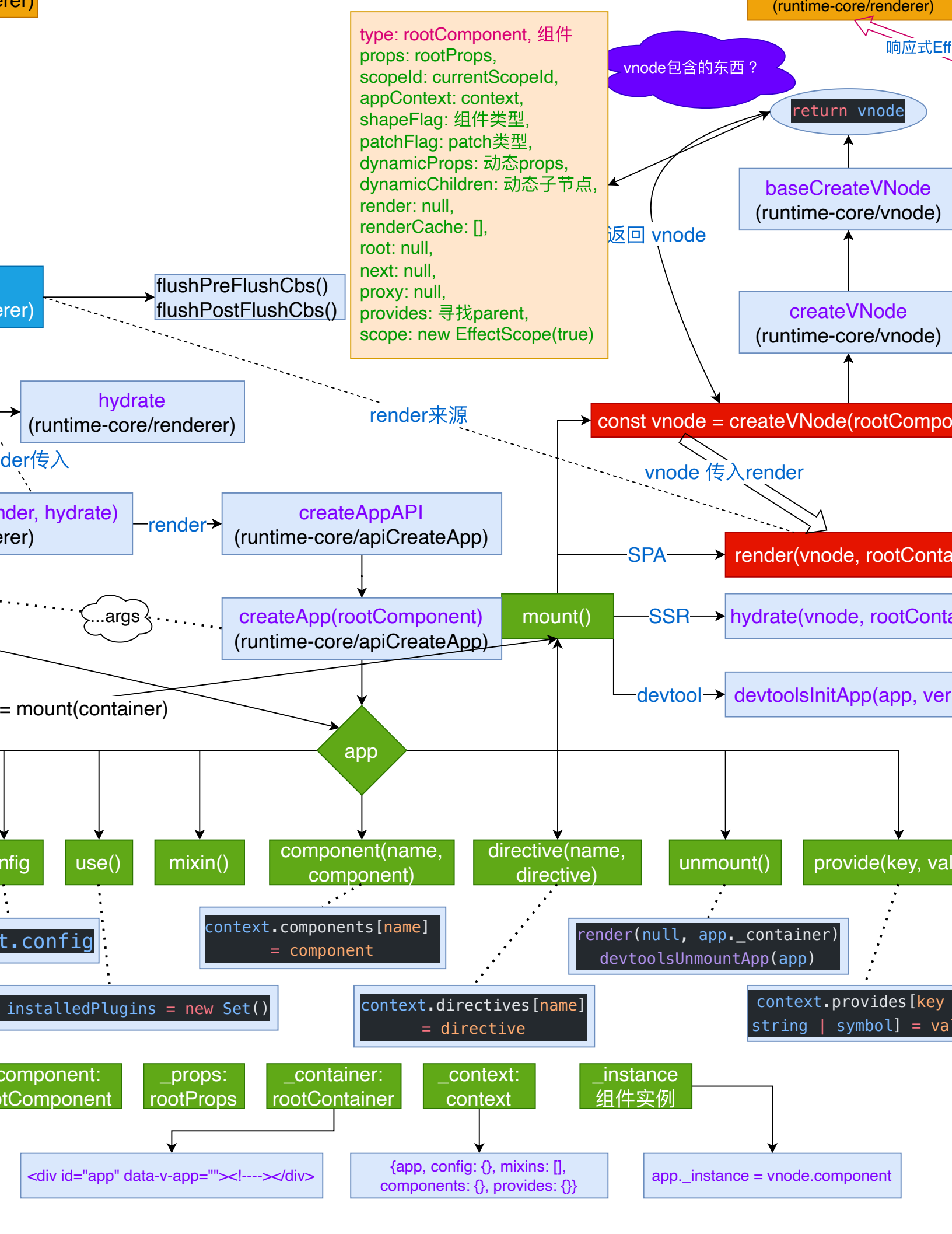

genHoists(ast.hoists, context)
(compiler-core/codegen)
判断type
const _directive_model =
_resloveDirective("model")
const _component_Comps =
_resolveComponent("Comps")
生成的代码
通过 ast.helpers的辅助函数导入
api，生成需要的 code
const _Vue = Vue
return function render(_ctx, _cache) {
with (_ctx) {
通过 ast.helpers.keys() 提供转换
const { renderList: _renderList, Fragment: _Fragment, openBlock: _openBlock,
createElementBlock: _createElementBlock, toDisplayString: _toDisplayString } = _Vue
return (_openBlock(), _createElementBlock("div", null, [
(_openBlock(true), _createElementBlock(_Fragment, null, _renderList(this.msg, (i)
=> {
return (_openBlock(), _createElementBlock("h3", { key: i.id },
_toDisplayString(i.id) + " -- " + _toDisplayString(i.title), 1 /* TEXT */))
}), 128 /* KEYED_FRAGMENT */))
]))
}
}
首次挂载 响应式数据变更时更新组件 递归执行 patch
Text
processText()
文本
1. hostInsert
2. hostSetText
Comment
processCommentNode()
1. hostInsert
2. hostCreateComment
Static
n1--null
mountStaticNode()
1. hostInsertStaticConent
__DEV__
patchStaticNode()
1. hostInsertStaticConent
2. hostNextSibing
3. removeStaticNode
Fragment
processFragment()
1. hostInsert
ELEMENT
processElement()
html 标签
mountElement()
挂载元素
patchElement()
patch 元素
mountChildren()
挂载子节点
dynamicChildren
patchBlockChildren()
动态节点 block树
diff
patchChildren()
patch 子节点 Diff
COMPONENT
processComponent()
组件
mountComponent()
组件首次渲染
updateComponent()
组件更新渲染
createComponentInstance()
创建组件实例 instance
(runtime-core/component)
setupComponent(instance)
执行 setup 函数
(runtime-core/component)
TELEPORT
type.process()
SUSPENSE
type.process()
setupRenderEffect()
监听副作用函数

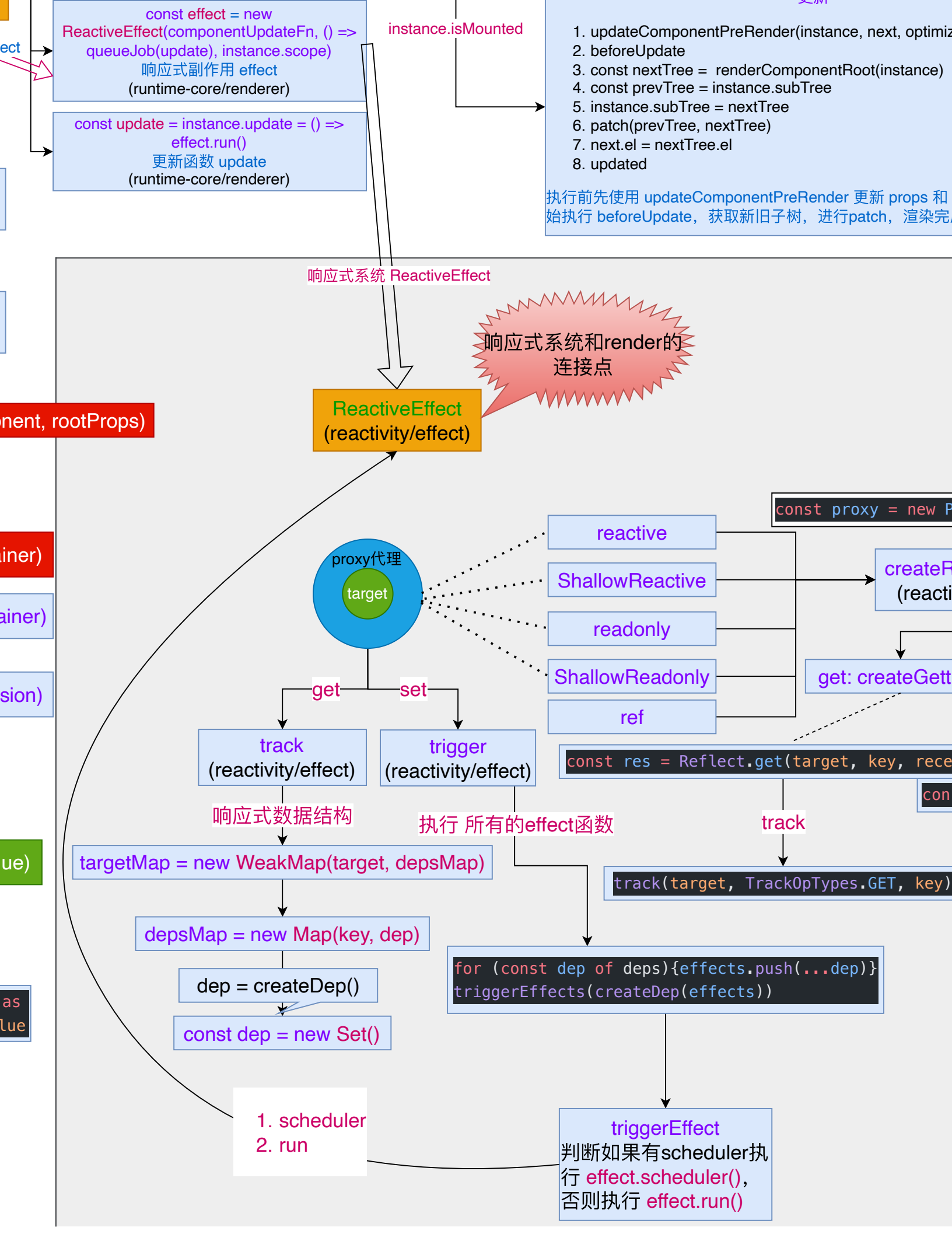

最快路径，有patchFlag
patchKeyedChildren()
patchFlag > 0
patchUnkeyedChildren()
没有 key
1. 获取最小长度 length
2. 遍历 patch
3. 判断 如果 oldLength > newLength
4. unmountChildren(c1) 移除旧的
5. 否则 挂载新的 mountChildren(c2)
子元素情况不同
新子元素是 文本
老子元素是 数组
老子元素是 文本
老子元素是 空
1. unmountChildren
2. 创建文本
设置文本
设置文本
新子元素是 空
老子元素是 数组
老子元素是 文本
老子元素是 空
unmountChildren
删除文本
不需要操作
新子元素是 数组
老子元素是 数组
老子元素是 文本
老子元素是 空
diff 操作
patchKeyedChildren()
mountChildren
mountChildren
initProps(instance, props)
初始化 props
(runtime-core/componentProps)
initSlots(instance, children)
初始化 slots
(runtime-core/componentSlots)
emit
emit 函数
(runtime-core/componentEmits)
instance.root
instance.emit
组件实例
const instance = {}
setupStatefulComponent
有状态组件的setup函数
(runtime-core/component)
组件实例 instance 包含的东西？
createSetupContext(instance)
创建setupContext，setup 函数第二个参数
(runtime-core/component)
return {
get attrs() {
return attrs || (at
},
slots: instance.slots
emit: instance.emit,
expose
}
handleSetupResult(instance, setupResult)
处理setup 的返回值
(runtime-core/component)
isFunction
isObject
instance.render = s
直接将setupResult赋
(直接返回渲染函数
instance.setupState
处理后的结果
执行完后执行下面
finishComponentSetup(instance)
完成组件setup操作，最后都会执行
(runtime-core/component)
const Component = instance.type
Compo
handle
挂载
1. beforeMount
2. const subTree = (instance.subTree = renderComponentRoot(instance))
3. mounted
先执行 beforeMount，subTree 为当前组件的子树，渲染完成后执行 mounted
! instance.isMounted
componentUpdateFn
组件更新函数
if()
更新

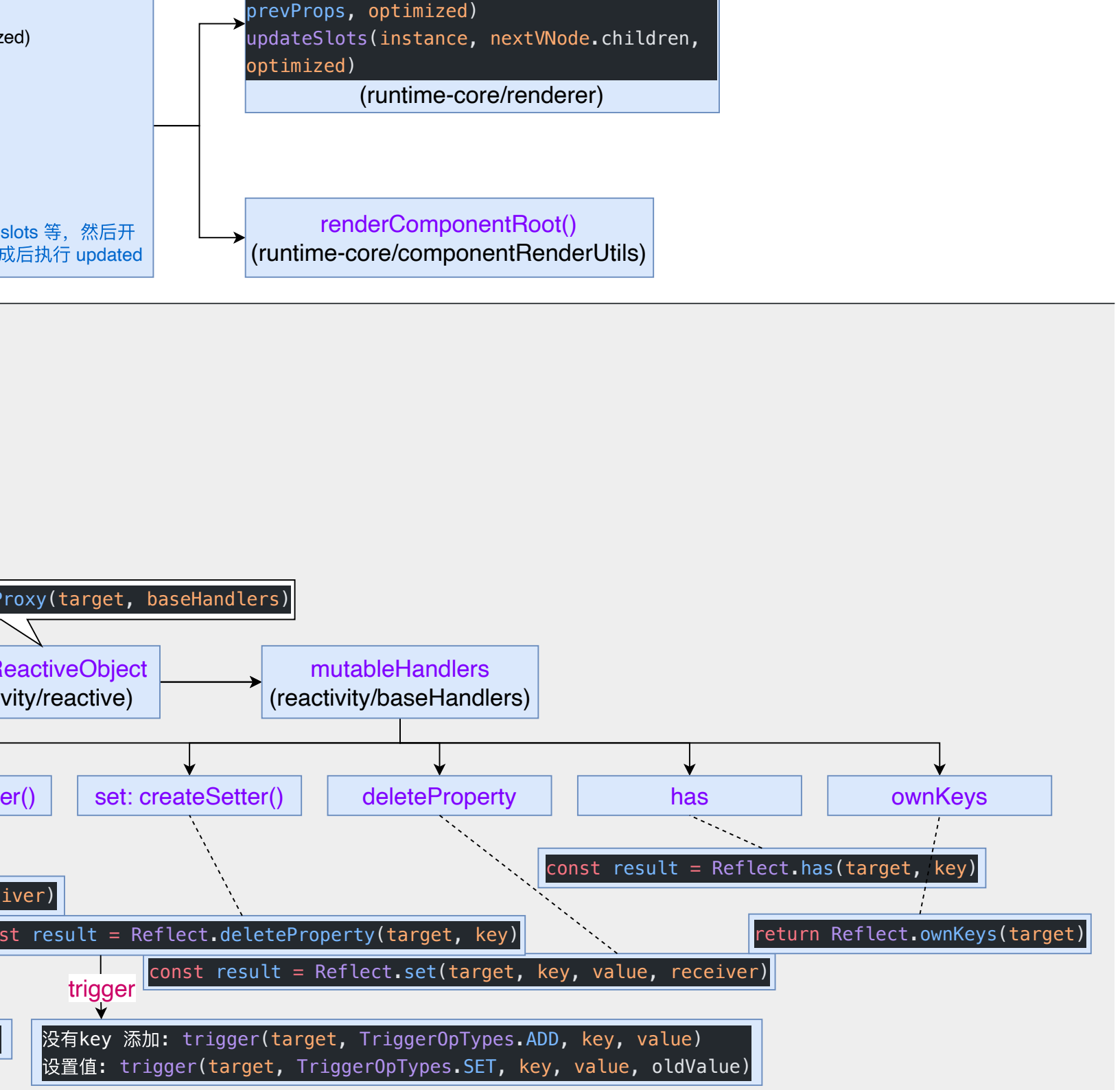

1. 相同的头部 patch
2. 相同的尾部 patch
3. 公共序列 和 挂载
4. 公共序列 和 卸载
5. 未知序列
trs = createAttrsProxy(instance))
createAttrsProxy(instance)
创建 attrs 的代理，触发 track
(runtime-core/component)
get(target, key: string) {
track(
instance,
TrackOpTypes.GET, '$attrs')
return target[key]
}
setupResult
值给render
的写法)
直接返回
new Proxy(objectWithRefs, shallowUnwrapHandlers)
shallowUnwrapHandlers
isReactive
使用 unref()
e = proxyRefs(setupResult)
果储存在 setupState 中
正常return {}
proxyRefs
在模版中去 .value 的处理
(reactivity/ref)
template 到编译器转
render 函数连接点
compile && !Component.render
template 存在
onent.render = compile(template,
finalCompilerOptions)
compile = compileToFunction
将编译器赋值，使compile为 true
(reactivity/ref)
registerRuntimeCompiler
注册运行时编译器
(runtime-core/component)
SetupResult(instance, setupResult)
处理setup 的返回值
(runtime-core/component)
instance.render = Component.render
获取到 render 函数
(runtime-core/component)
updateComponentPreRender
updateProps(instance, nextVNode.props,


(compiler-core/parse)
attr = parseAttribute()
v-xx bind on slot
(compiler-core/parse)
{
setup() {},
render() {}
}
编写入口
...args
const app = ensureRenderer().createApp(...args)
const app = createApp(App)
createApp
(runtime-dom/index)
ensureRenderer
(runtime-dom/index)
ensureRenderer 调用返回 createApp
createApp是createAppAPI 调用的返回
renderer = createRenderer(rende
参数
1. nodeOps
2. patchProp
insert()
插入元素
remove()
移除元素
createElement()
创建元素
createText()
创建文本
createComment()
创建注释
setText()
设置文本
patchClass()
类名
setText()
设置文本
createElementText()
设置元素内容
insertStaticContent()
设置静态内容
setScopeId()
设置ScopeId
querySelector()
查询选择器
patchDOM
prop
const proxy


卸载老的vnode
unmount
卸载
vnode==null
render
(runtime-core/rende
创建渲染器
rendererOptions
createRenderer
(runtime-core/renderer)
options
baseCreateRenderer
(runtime-core/renderer)
return
ren
createApp: createAppAPI(ren
(runtime-core/rende
解构传入的options
返回app
erOptions)
const {
insert: hostInsert,
remove: hostRemove,
patchProp: hostPatchProp,
createElement: hostCreateElement,
createText: hostCreateText,
createComment: hostCreateComment,
setText: hostSetText,
setElementText: hostSetElementText,
parentNode: hostParentNode,
nextSibling: hostNextSibling,
setScopeId: hostSetScopeId = NOOP,
cloneNode: hostCloneNode,
insertStaticContent: hostInsertStaticContent
} = options
const proxy
patchStyle()
样式
patchEvent()
事件
MProp()
o
patchAttr()
attr
version
co
pkg.version
contex
const
= app.mount('#app')
app.mount: (containerOrSelector) => {}
(runtime-dom/index)
_uid: uid++
roo
return getExposeProxy(vnode.component!)
|| vnode.component!.proxy
app 包含的东西？


(runtime-core/renderer)
响应式Eff
type: rootComponent, 组件
props: rootProps,
scopeId: currentScopeId,
appContext: context,
shapeFlag: 组件类型,
patchFlag: patch类型,
dynamicProps: 动态props,
dynamicChildren: 动态子节点,
render: null,
renderCache: [],
root: null,
next: null,
proxy: null,
provides: 寻找parent,
scope: new EffectScope(true)
vnode包含的东西？
return vnode
baseCreateVNode
(runtime-core/vnode)
返回 vnode
createVNode
(runtime-core/vnode)
flushPreFlushCbs()
flushPostFlushCbs()
hydrate
(runtime-core/renderer)
render来源
const vnode = createVNode(rootCompo
vnode 传入render
render(vnode, rootConta
SPA
createAppAPI
(runtime-core/apiCreateApp)
render
createApp(rootComponent)
(runtime-core/apiCreateApp)
...args
mount()
SSR
hydrate(vnode, rootConta
devtool
devtoolsInitApp(app, ver
= mount(container)
app
use()
mixin()
component(name, component)
directive(name, directive)
unmount()
provide(key, val
t.config
context.components[name] = component
render(null, app._container)
devtoolsUnmountApp(app)
installedPlugins = new Set()
context.directives[name] = directive
context.provides[key
string | symbol] = va
component:
tComponent
_props:
rootProps
_container:
rootContainer
_context:
context
_instance
组件实例
<!---->
{app, config: {}, mixins: [], components: {}, provides: {}}
app._instance = vnode.component


const effect = new ReactiveEffect(componentUpdateFn, () => queueJob(update), instance.scope)
响应式副作用 effect
(runtime-core/renderer)
const update = instance.update = () => effect.run()
更新函数 update
(runtime-core/renderer)
instance.isMounted
1. updateComponentPreRender(instance, next, optimiz
2. beforeUpdate
3. const nextTree = renderComponentRoot(instance)
4. const prevTree = instance.subTree
5. instance.subTree = nextTree
6. patch(prevTree, nextTree)
7. next.el = nextTree.el
8. updated
执行前先使用 updateComponentPreRender 更新 props 和
始执行 beforeUpdate，获取新旧子树，进行patch，渲染完
响应式系统 ReactiveEffect
响应式系统和render的连接点
ReactiveEffect
(reactivity/effect)
proxy代理
target
reactive
ShallowReactive
readonly
ShallowReadonly
ref
const proxy = new P
createR
(reacti
get: createGett
get
set
track
(reactivity/effect)
trigger
(reactivity/effect)
const res = Reflect.get(target, key, rece
con
track
track(target, TrackOpTypes.GET, key)
响应式数据结构
执行 所有的effect函数
targetMap = new WeakMap(target, depsMap)
depsMap = new Map(key, dep)
dep = createDep()
const dep = new Set()
for (const dep of deps){effects.push(...dep)}
triggerEffects(createDep(effects))
1. scheduler
2. run
triggerEffect
判断如果有scheduler执行 effect.scheduler()，
否则执行 effect.run()


zed)
prevProps, optimized)
updateSlots(instance, nextVNode.children,
optimized)
(runtime-core/renderer)
slots 等，然后开
成后执行 updated
renderComponentRoot()
(runtime-core/componentRenderUtils)
Proxy(target, baseHandlers)
ReactiveObject
vity/reactive)
mutableHandlers
(reactivity/baseHandlers)
er()
set: createSetter()
deleteProperty
has
ownKeys
const result = Reflect.has(target, key)
iver)
st result = Reflect.deleteProperty(target, key)
return Reflect.ownKeys(target)
const result = Reflect.set(target, key, value, receiver)
trigger
没有key 添加: trigger(target, TriggerOpTypes.ADD, key, value)
设置值: trigger(target, TriggerOpTypes.SET, key, value, oldValue)