在 Vue.js 中，组件是一个非常重要的概念，整个应用的页面都是通过组件渲染来实现的，但是你知道当我们编写这些组件的时候，它的内部是如何工作的吗？从我们编写组件开始，到最终真实的 DOM 又是怎样的一个转变过程呢？这节课，我们将会学习 Vue.js 3.0 中的组件是如何渲染的，通过学习，你的这些问题将会迎刃而解。

首先，组件是一个抽象的概念，它是对一棵 DOM 树的抽象，我们在页面中写一个组件节点：

```
<hello-world></hello-world>
```

这段代码并不会在页面上渲染一个 `<hello-world>` 标签，而它具体渲染成什么，取决于你怎么编写 HelloWorld 组件的模板。举个例子，HelloWorld 组件内部的模板定义是这样的：

```
<template>

<div>

<p>Hello World</p>

</div>

</template>
```

可以看到，模板内部最终会在页面上渲染一个 div，内部包含一个 p 标签，用来显示 Hello World 文本。

所以，从表现上来看，组件的模板决定了组件生成的 DOM 标签，而在 Vue.js 内部，一个组件想要真正的渲染生成 DOM，还需要经历 “创建 vnode - 渲染 vnode - 生成 DOM” 这几个步骤：

你可能会问，什么是 vnode，它和组件什么关系呢？先不要着急，我们在后面会详细说明。这里，你只需要记住它就是一个可以描述组件信息的 JavaScript 对象即可。

接下来，我们就从应用程序的入口开始，逐步来看 Vue.js 3.0 中的组件是如何渲染的。

应用程序初始化

一个组件可以通过 “模板加对象描述” 的方式创建，组件创建好以后是如何被调用并初始化的呢？因为整个组件树是由根组件开始渲染的，为了找到根组件的渲染入口，我们需要从应用程序的初始化过程开始分析。

在这里，我分别给出了通过 Vue.js 2.x 和 Vue.js 3.0 来初始化应用的代码：

```
import Vue from 'vue'
```

```
import App from './App'

const app = new Vue({

  render: h => h(App)

})

app.$mount('#app')
```

```
import { createApp } from 'vue'

import App from './app'

const app = createApp(App)

app.mount('#app')
```

可以看到，Vue.js 3.0 初始化应用的方式和 Vue.js 2.x 差别并不大，本质上都是把 App 组件挂载到 id 为 app 的 DOM 节点上。

但是，在 Vue.js 3.0 中还导入了一个 createApp，其实这是个入口函数，它是 Vue.js 对外暴露的一个函数，我们来看一下它的内部实现：

```
const createApp = ((...args) => {

const app = ensureRenderer().createApp(...args)

const { mount } = app

  app.mount = (containerOrSelector) => {

  }

return app

})
```

从代码中可以看出 createApp 主要做了两件事情：创建 app 对象和重写 app.mount 方法。接下来，我们就具体来分析一下它们。

1. 创建 app 对象

首先，我们使用 ensureRenderer().createApp() 来创建 app 对象：

```
const app = ensureRenderer().createApp(...args)
```

其中 ensureRenderer() 用来创建一个渲染器对象，它的内部代码是这样的：

```
const rendererOptions = {

  patchProp,

  ...nodeOps

}

let renderer

function ensureRenderer() {

return renderer || (renderer = createRenderer(rendererOptions))

}

function createRenderer(options) {

return baseCreateRenderer(options)

}

function baseCreateRenderer(options) {

function render(vnode, container) {

  }

return {
```

```
    render,

    createApp: createAppAPI(render)

  }

}

function createAppAPI(render) {

return function createApp(rootComponent, rootProps = null) {

const app = {

      _component: rootComponent,

      _props: rootProps,

      mount(rootContainer) {

const vnode = createVNode(rootComponent, rootProps)

        render(vnode, rootContainer)

        app._container = rootContainer

return vnode.component.proxy

      }

    }

return app

  }

}
```

可以看到，这里先用 ensureRenderer() 来延时创建渲染器，这样做的好处是当用户只依赖响应式包的时候，就不会创建渲染器，因此可以通过 tree-shaking 的方式移除核心渲染逻辑相关的代码。

这里涉及了渲染器的概念，它是为跨平台渲染做准备的，之后我会在自定义渲染器的相关内容中详细说明。在这里，你可以简单地把渲染器理解为包含平台渲染核心逻辑的 JavaScript 对象。

我们结合上面的代码继续深入，在 Vue.js 3.0 内部通过 createRenderer 创建一个渲染器，这个渲染器内部会有一个 createApp 方法，它是执行 createAppAPI 方法返回的函数，接受了 rootComponent 和 rootProps 两个参数，我们在应用层面执行 createApp(App) 方法时，会把 App 组件对象作为根组件传递给 rootComponent。这样，createApp 内部就创建了一个 app 对象，它会提供 mount 方法，这个方法是用来挂载组件的。

在整个 app 对象创建过程中，Vue.js 利用闭包和函数柯里化的技巧，很好地实现了参数保留。比如，在执行 app.mount 的时候，并不需要传入渲染器 render，这是因为在执行 createAppAPI 的时候渲染器 render 参数已经被保留下来了。

2. 重写 app.mount 方法

接下来，是重写 app.mount 方法。

根据前面的分析，我们知道 createApp 返回的 app 对象已经拥有了 mount 方法了，但在入口函数中，接下来的逻辑却是对 app.mount 方法的重写。先思考一下，为什么要重写这个方法，而不把相关逻辑放在 app 对象的 mount 方法内部来实现呢?

这是因为 Vue.js 不仅仅是为 Web 平台服务，它的目标是支持跨平台渲染，而 createApp 函数内部的 app.mount 方法是一个标准的可跨平台的组件渲染流程：

```
mount(rootContainer) {

const vnode = createVNode(rootComponent, rootProps)

  render(vnode, rootContainer)

  app._container = rootContainer

return vnode.component.proxy

}
```

标准的跨平台渲染流程是先创建 vnode，再渲染 vnode。此外参数 rootContainer 也可以是不同类型的值，比如，在 Web 平台它是一个 DOM 对象，而在其他平台（比如 Weex 和小程序）中可以是其他类型的值。所以这里面的代码不应该包含任何特定平台相关的逻辑，也就是说这些代码的执行逻辑都是与平台无关的。因此我们需要在外部重写这个方法，来完善 Web 平台下的渲染逻辑。

接下来，我们再来看 app.mount 重写都做了哪些事情：

```
app.mount = (containerOrSelector) => {

const container = normalizeContainer(containerOrSelector)
```

```
if (!container)

return

const component = app._component

if (!isFunction(component) && !component.render && !component.template) {

    component.template = container.innerHTML

  }

  container.innerHTML = ''

return mount(container)

}
```

首先是通过 normalizeContainer 标准化容器（这里可以传字符串选择器或者 DOM 对象，但如果是字符串选择器，就需要把它转成 DOM 对象，作为最终挂载的容器），然后做一个 if 判断，如果组件对象没有定义 render 函数和 template 模板，则取容器的 innerHTML 作为组件模板内容；接着在挂载前清空容器内容，最终再调用 app.mount 的方法走标准的组件渲染流程。

在这里，重写的逻辑都是和 Web 平台相关的，所以要放在外部实现。此外，这么做的目的是既能让用户在使用 API 时可以更加灵活，也兼容了 Vue.js 2.x 的写法，比如 app.mount 的第一个参数就同时支持选择器字符串和 DOM 对象两种类型。

从 app.mount 开始，才算真正进入组件渲染流程，那么接下来，我们就重点看一下核心渲染流程做的两件事情：创建 vnode 和渲染 vnode。

核心渲染流程：创建 vnode 和渲染 vnode

1. 创建 vnode

首先，是创建 vnode 的过程。

vnode 本质上是用来描述 DOM 的 JavaScript 对象，它在 Vue.js 中可以描述不同类型的节点，比如普通元素节点、组件节点等。

什么是**普通元素节点**呢？举个例子，在 HTML 中我们使用 `<button>` 标签来写一个按钮：

```
<button>click me</button>
```

我们可以用 vnode 这样表示 `<button>` 标签：

```
const vnode = {
```

```
  type: 'button',
  props: {
'class': 'btn',
    style: {
      width: '100px',
      height: '50px'
    }
  },
  children: 'click me'
}
```

其中，type 属性表示 DOM 的标签类型，props 属性表示 DOM 的一些附加信息，比如 style 、class 等，children 属性表示 DOM 的子节点，它也可以是一个 vnode 数组，只不过 vnode 可以用字符串表示简单的文本 。

什么是**组件节点**呢？其实， vnode 除了可以像上面那样用于描述一个真实的 DOM，也可以用来描述组件。

我们先在模板中引入一个组件标签 `<custom-component>`：

```
<custom-component msg="test"></custom-component>
```

我们可以用 vnode 这样表示 `<custom-component>` 组件标签：

```
const CustomComponent = {
}
const vnode = {
  type: CustomComponent,
```

```
  props: {
    msg: 'test'
  }
}
```

组件 vnode 其实是**对抽象事物的描述**，这是因为我们并不会在页面上真正渲染一个 `<custom-component>` 标签，而是渲染组件内部定义的 HTML 标签。

除了上两种 vnode 类型外，还有纯文本 vnode、注释 vnode 等等，但鉴于我们的主线只需要研究组件 vnode 和普通元素 vnode，所以我在这里就不赘述了。

另外，Vue.js 3.0 内部还针对 vnode 的 type，做了更详尽的分类，包括 Suspense、Teleport 等，且把 vnode 的类型信息做了编码，以便在后面的 patch 阶段，可以根据不同的类型执行相应的处理逻辑：

```
const shapeFlag = isString(type)
  ? 1
  : isSuspense(type)
    ? 128
    : isTeleport(type)
      ? 64
      : isObject(type)
        ? 4
        : isFunction(type)
          ? 2
          : 0
```

知道什么是 vnode 后，你可能会好奇，那么 vnode 有什么优势呢？为什么一定要设计 vnode 这样的数据结构呢？

首先是**抽象**，引入 vnode，可以把渲染过程抽象化，从而使得组件的抽象能力也得到提升。

其次是**跨平台**，因为 patch vnode 的过程不同平台可以有自己的实现，基于 vnode 再做服务端渲染、Weex 平台、小程序平台的渲染都变得容易了很多。

不过这里要特别注意，使用 vnode 并不意味着不用操作 DOM 了，很多同学会误以为 vnode 的性能一定比手动操作原生 DOM 好，这个其实是不一定的。

因为，首先这种基于 vnode 实现的 MVVM 框架，在每次 render to vnode 的过程中，渲染组件会有一定的 JavaScript 耗时，特别是大组件，比如一个 1000 * 10 的 Table 组件，render to vnode 的过程会遍历 1000 * 10 次去创建内部 cell vnode，整个耗时就会变得比较长，加上 patch vnode 的过程也会有一定的耗时，当我们去更新组件的时候，用户会感觉到明显的卡顿。虽然 diff 算法在减少 DOM 操作方面足够优秀，但最终还是免不了操作 DOM，所以说性能并不是 vnode 的优势。

那么，Vue.js 内部是如何创建这些 vnode 的呢？

回顾 app.mount 函数的实现，内部是通过 createVNode 函数创建了根组件的 vnode：

```
const vnode = createVNode(rootComponent, rootProps)
```

我们来看一下 createVNode 函数的大致实现：

```
function createVNode(type, props = null

,children = null) {

if (props) {

  }

const shapeFlag = isString(type)

    ? 1

    : isSuspense(type)

      ? 128

      : isTeleport(type)

        ? 64

        : isObject(type)
```

```
          ? 4

          : isFunction(type)

            ? 2

            : 0

const vnode = {

    type,

    props,

    shapeFlag,

  }

  normalizeChildren(vnode, children)

return vnode

}
```

通过上述代码可以看到，其实 createVNode 做的事情很简单，就是：对 props 做标准化处理、对 vnode 的类型信息编码、创建 vnode 对象，标准化子节点 children 。

我们现在拥有了这个 vnode 对象，接下来要做的事情就是把它渲染到页面中去。

2. 渲染 vnode

接下来，是渲染 vnode 的过程。

回顾 app.mount 函数的实现，内部通过执行这段代码去渲染创建好的 vnode：

```
render(vnode, rootContainer)

const render = (vnode, container) => {

if (vnode == null) {

if (container._vnode) {
```

```
      unmount(container._vnode, null, null, true)

    }

  } else {

    patch(container._vnode || null, vnode, container)

  }

  container._vnode = vnode

}
```

这个渲染函数 render 的实现很简单，如果它的第一个参数 vnode 为空，则执行销毁组件的逻辑，否则执行创建或者更新组件的逻辑。

接下来我们接着看一下上面渲染 vnode 的代码中涉及的 patch 函数的实现：

```
const patch = (n1, n2, container, anchor = null, parentComponent = null,
parentSuspense = null, isSVG = false, optimized = false) => {

if (n1 && !isSameVNodeType(n1, n2)) {

    anchor = getNextHostNode(n1)

    unmount(n1, parentComponent, parentSuspense, true)

    n1 = null

  }

const { type, shapeFlag } = n2

switch (type) {

case Text:

break
```

```
case Comment:

break

case Static:

break

case Fragment:

break

default:

if (shapeFlag & 1 ) {

        processElement(n1, n2, container, anchor, parentComponent,
parentSuspense, isSVG, optimized)

      }

else if (shapeFlag & 6 ) {

        processComponent(n1, n2, container, anchor, parentComponent,
parentSuspense, isSVG, optimized)

      }

else if (shapeFlag & 64 ) {

      }

else if (shapeFlag & 128 ) {

      }

  }
```

```
}
```

patch 本意是打补丁的意思，这个函数有两个功能，一个是根据 vnode 挂载 DOM，一个是根据新旧 vnode 更新 DOM。对于初次渲染，我们这里只分析创建过程，更新过程在后面的章节分析。

在创建的过程中，patch 函数接受多个参数，这里我们目前只重点关注前三个：

1. 第一个参数 **n1 表示旧的 vnode**，当 n1 为 null 的时候，表示是一次挂载的过程；
2. 第二个参数 **n2 表示新的 vnode 节点**，后续会根据这个 vnode 类型执行不同的处理逻辑；
3. 第三个参数 **container 表示 DOM 容器**，也就是 vnode 渲染生成 DOM 后，会挂载到 container 下面。

对于渲染的节点，我们这里重点关注两种类型节点的渲染逻辑：对组件的处理和对普通 DOM 元素的处理。

先来看对组件的处理。由于初始化渲染的是 App 组件，它是一个组件 vnode，所以我们来看一下组件的处理逻辑是怎样的。首先是用来处理组件的 processComponent 函数的实现：

```
const processComponent = (n1, n2, container, anchor, parentComponent,
parentSuspense, isSVG, optimized) => {


if (n1 == null) {


   mountComponent(n2, container, anchor, parentComponent, parentSuspense, isSVG,
optimized)


  }


else {


    updateComponent(n1, n2, parentComponent, optimized)


  }


}
```

该函数的逻辑很简单，如果 n1 为 null，则执行挂载组件的逻辑，否则执行更新组件的逻辑。

我们接着来看挂载组件的 mountComponent 函数的实现：

```
const mountComponent = (initialVNode, container, anchor, parentComponent,
parentSuspense, isSVG, optimized) => {


const instance = (initialVNode.component = createComponentInstance(initialVNode,
parentComponent, parentSuspense))
```

```
  setupComponent(instance)


  setupRenderEffect(instance, initialVNode, container, anchor, parentSuspense,
isSVG, optimized)


}
```

可以看到，挂载组件函数 mountComponent 主要做三件事情：创建组件实例、设置组件实例、设置并运行带副作用的渲染函数。

首先是创建组件实例，Vue.js 3.0 虽然不像 Vue.js 2.x 那样通过类的方式去实例化组件，但内部也通过对象的方式去创建了当前渲染的组件实例。

其次设置组件实例，instance 保留了很多组件相关的数据，维护了组件的上下文，包括对 props、插槽，以及其他实例的属性的初始化处理。

创建和设置组件实例这两个流程我们这里不展开讲，会在后面的章节详细分析。

最后是运行带副作用的渲染函数 setupRenderEffect，我们重点来看一下这个函数的实现：

```
const setupRenderEffect = (instance, initialVNode, container, anchor,
parentSuspense, isSVG, optimized) => {


  instance.update = effect(function componentEffect() {


if (!instance.isMounted) {


const subTree = (instance.subTree = renderComponentRoot(instance))


      patch(null, subTree, container, anchor, instance, parentSuspense, isSVG)


      initialVNode.el = subTree.el


      instance.isMounted = true


    }


else {


    }


  }, prodEffectOptions)


}
```

该函数利用响应式库的 effect 函数创建了一个副作用渲染函数 componentEffect （effect 的实现我们后面讲响应式章节会具体说）。**副作用**，这里你可以简单地理解为，当组件的数据发生变化时，effect 函数包裹的内部渲染函数 componentEffect 会重新执行一遍，从而达到重新渲染组件的目的。

渲染函数内部也会判断这是一次初始渲染还是组件更新。这里我们只分析初始渲染流程。

初始渲染主要做两件事情：渲染组件生成 subTree、把 subTree 挂载到 container 中。

首先，是渲染组件生成 subTree，它也是一个 vnode 对象。这里要注意别把 subTree 和 initialVNode 弄混了（其实在 Vue.js 3.0 中，根据命名我们已经能很好地区分它们了，而在 Vue.js 2.x 中它们分别命名为 _vnode 和 $vnode）。我来举个例子说明，在父组件 App 中里引入了 Hello 组件：

```
<template>

<div>

<p>This is an app.</p>

<hello></hello>

</div>

</template>
```

在 Hello 组件中是 `<div>` 标签包裹着一个 `<p>` 标签：

```
<template>

<div>

<p>Hello, Vue 3.0!</p>

</div>

</template>
```

在 App 组件中， `<hello>` 节点渲染生成的 vnode ，对应的就是 Hello 组件的 initialVNode ，为了好记，你也可以把它称作 “组件 vnode”。而 Hello 组件内部整个 DOM 节点对应的 vnode 就是执行 renderComponentRoot 渲染生成对应的 subTree，我们可以把它称作 “子树 vnode”。

我们知道每个组件都会有对应的 render 函数，即使你写 template，也会编译成 render 函数，而 renderComponentRoot 函数就是去执行 render 函数创建整个组件树内部的 vnode，把这个 vnode 再经过内部一层标准化，就得到了该函数的返回结果：子树 vnode。

渲染生成子树 vnode 后，接下来就是继续调用 patch 函数把子树 vnode 挂载到 container 中了。

那么我们又再次回到了 patch 函数，会继续对这个子树 vnode 类型进行判断，对于上述例子，App 组件的根节点是 `<div>` 标签，那么对应的子树 vnode 也是一个普通元素 vnode，那么我们**接下来看对普通 DOM 元素的处理流程。**

首先我们来看一下处理普通 DOM 元素的 processElement 函数的实现：

```
const processElement = (n1, n2, container, anchor, parentComponent,
parentSuspense, isSVG, optimized) => {


  isSVG = isSVG || n2.type === 'svg'


if (n1 == null) {


    mountElement(n2, container, anchor, parentComponent, parentSuspense, isSVG,
optimized)


  }


else {


    patchElement(n1, n2, parentComponent, parentSuspense, isSVG, optimized)


  }


}
```

该函数的逻辑很简单，如果 n1 为 null，走挂载元素节点的逻辑，否则走更新元素节点逻辑。

我们接着来看挂载元素的 mountElement 函数的实现：

```
const mountElement = (vnode, container, anchor, parentComponent, parentSuspense,
isSVG, optimized) => {


  let el


const { type, props, shapeFlag } = vnode


  el = vnode.el = hostCreateElement(vnode.type, isSVG, props && props.is)


if (props) {
```

```
for (const key in props) {

if (!isReservedProp(key)) {

        hostPatchProp(el, key, null, props[key], isSVG)

      }

    }

  }

if (shapeFlag & 8 ) {

    hostSetElementText(el, vnode.children)

  }

else if (shapeFlag & 16 ) {

    mountChildren(vnode.children, el, null, parentComponent, parentSuspense,
isSVG && type !== 'foreignObject', optimized || !!vnode.dynamicChildren)

  }

  hostInsert(el, container, anchor)

}
```

可以看到，挂载元素函数主要做四件事：创建 DOM 元素节点、处理 props、处理 children、挂载 DOM 元素到 container 上。

首先是创建 DOM 元素节点，通过 hostCreateElement 方法创建，这是一个平台相关的方法，我们来看一下它在 Web 环境下的定义：

```
function createElement(tag, isSVG, is) {

  isSVG ? document.createElementNS(svgNS, tag)

    : document.createElement(tag, is ? { is } : undefined)

}
```

它调用了底层的 DOM API document.createElement 创建元素，所以本质上 Vue.js 强调不去操作 DOM ，只是希望用户不直接碰触 DOM，它并没有什么神奇的魔法，底层还是会操作 DOM。

另外，如果是其他平台比如 Weex，hostCreateElement 方法就不再是操作 DOM ，而是平台相关的 API 了，这些平台相关的方法是在创建渲染器阶段作为参数传入的。

创建完 DOM 节点后，接下来要做的是判断如果有 props 的话，给这个 DOM 节点添加相关的 class、style、event 等属性，并做相关的处理，这些逻辑都是在 hostPatchProp 函数内部做的，这里就不展开讲了。

接下来是对子节点的处理，我们知道 DOM 是一棵树，vnode 同样也是一棵树，并且它和 DOM 结构是一一映射的。

如果子节点是纯文本，则执行 hostSetElementText 方法，它在 Web 环境下通过设置 DOM 元素的 textContent 属性设置文本：

```
function setElementText(el, text) {

  el.textContent = text

}
```

如果子节点是数组，则执行 mountChildren 方法：

```
const mountChildren = (children, container, anchor, parentComponent,
parentSuspense, isSVG, optimized, start = 0) => {

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

const child = (children[i] = optimized

      ? cloneIfMounted(children[i])

      : normalizeVNode(children[i]))
```

```
    patch(null, child, container, anchor, parentComponent, parentSuspense,
isSVG, optimized)

  }

}
```

子节点的挂载逻辑同样很简单，遍历 children 获取到每一个 child，然后递归执行 patch 方法挂载每一个 child 。注意，这里有对 child 做预处理的情况（后面编译优化的章节会详细分析）。

可以看到，mountChildren 函数的第二个参数是 container，而我们调用 mountChildren 方法传入的第二个参数是在 mountElement 时创建的 DOM 节点，这就很好地建立了父子关系。

另外，通过递归 patch 这种深度优先遍历树的方式，我们就可以构造完整的 DOM 树，完成组件的渲染。

处理完所有子节点后，最后通过 hostInsert 方法把创建的 DOM 元素节点挂载到 container 上，它在 Web 环境下这样定义：

```
function insert(child, parent, anchor) {

if (anchor) {

    parent.insertBefore(child, anchor)

  }

else {

    parent.appendChild(child)

  }

}
```

这里会做一个 if 判断，如果有参考元素 anchor，就执行 parent.insertBefore ，否则执行 parent.appendChild 来把 child 添加到 parent 下，完成节点的挂载。

因为 insert 的执行是在处理子节点后，所以挂载的顺序是先子节点，后父节点，最终挂载到最外层的容器上。

> **知识延伸：嵌套组件**
> 细心的你可能会发现，在 mountChildren 的时候递归执行的是 patch 函数，而不是 mountElement 函数，这是因为子节点可能有其他类型的 vnode，比如组件 vnode。

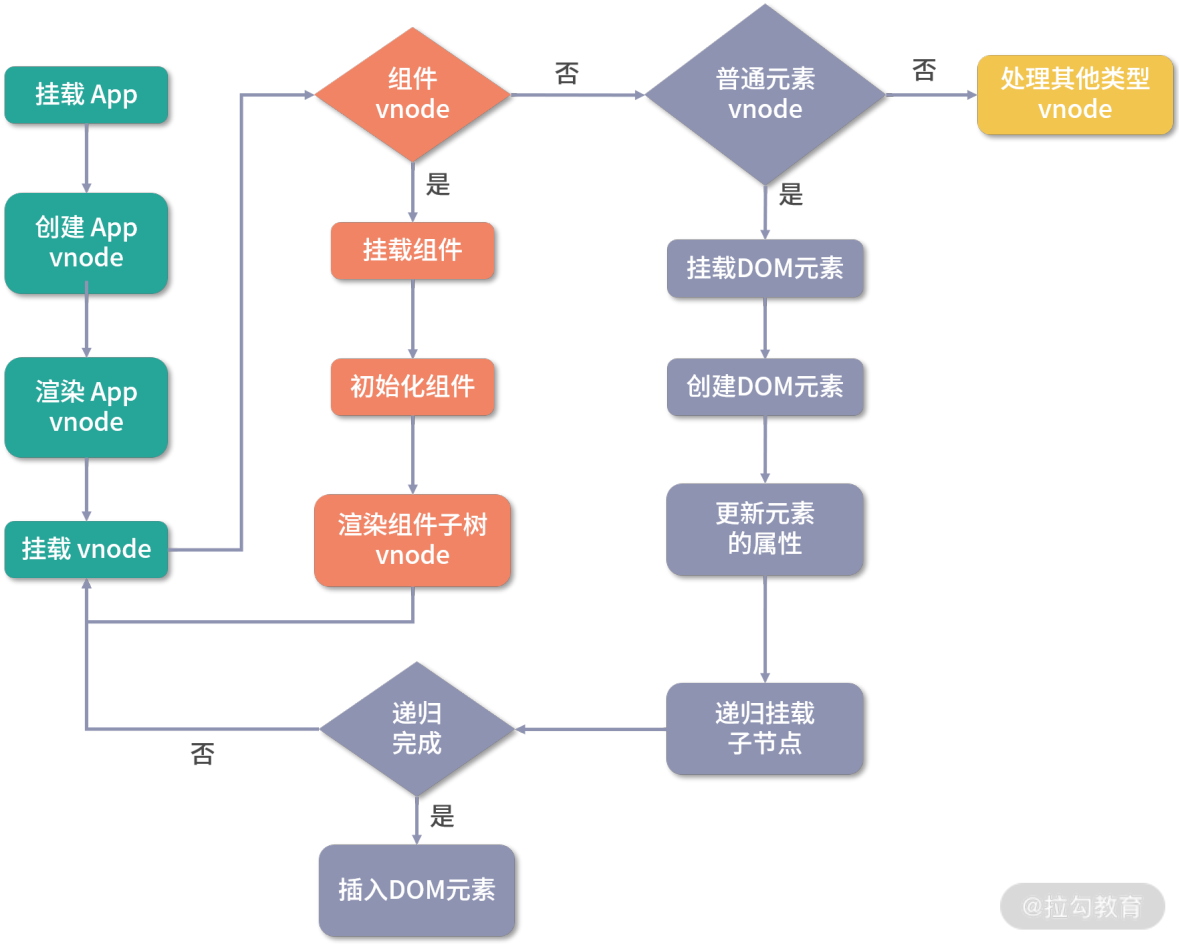
> 在真实开发场景中，嵌套组件场景是再正常不过的了，前面我们举的 App 和 Hello 组件的例子就是嵌套组件的场景。组件 vnode 主要维护着组件的定义对象，组件上的各种 props，而组件本身是一个抽象节点，它自身的渲染其实是通过执行组件定义的 render 函数渲染生成的子树 vnode 来完成，然后再 patch 。通过这种递归的方式，无论组件的嵌套层级多深，都可以完成整个组件树的渲染。

总结

OK，到这里我们这一节的学习也要结束啦，这节课我们主要分析了组件的渲染流程，从入口开始，一层层分析组件渲染。

你可能发现了，文中提到的很多技术点我会放在后面的章节去讲，这样做是为了让我们不跑题，重点放在理解组件的渲染流程上。下节课我将会带你具体分析一下组件的更新过程。

这里，我用一张图来带你更加直观地感受下整个组件渲染流程：



最后，给你留一道思考题目，我们平时开发页面就是把页面拆成一个个组件，那么组件的拆分粒度是越细越好吗？为什么呢？欢迎你在留言区与我分享。

> **本节课的相关代码在源代码中的位置如下：**
> packages/runtime-dom/src/index.ts
> packages/runtime-core/src/apiCreateApp.ts
> packages/runtime-core/src/vnode.ts
> packages/runtime-core/src/renderer.ts
> packages/runtime-dom/src/nodeOps.ts