上一节课我们梳理了组件渲染的过程，本质上就是把各种类型的 vnode 渲染成真实 DOM。我们也知道了组件是由模板、组件描述对象和数据构成的，数据的变化会影响组件的变化。组件的渲染过程中创建了一个带副作用的渲染函数，当数据变化的时候就会执行这个渲染函数来触发组件的更新。那么接下来，我们就具体分析一下组件的更新过程。

副作用渲染函数更新组件的过程

我们先来回顾一下带副作用渲染函数 setupRenderEffect 的实现，但是这次我们要重点关注更新组件部分的逻辑：

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


  instance.update = effect(function componentEffect() {


if (!instance.isMounted) {


    }


else {


      let { next, vnode } = instance


if (next) {


        updateComponentPreRender(instance, next, optimized)


      }


else {


        next = vnode


      }


const nextTree = renderComponentRoot(instance)


const prevTree = instance.subTree
```

```
      instance.subTree = nextTree

      patch(prevTree, nextTree,
        hostParentNode(prevTree.el),
        getNextHostNode(prevTree),
        instance,
        parentSuspense,
        isSVG)
      next.el = nextTree.el
    }
  }, prodEffectOptions)
}
```

可以看到，更新组件主要做三件事情：**更新组件 vnode 节点、渲染新的子树 vnode、根据新旧子树 vnode 执行 patch 逻辑。**

首先是更新组件 vnode 节点，这里会有一个条件判断，判断组件实例中是否有新的组件 vnode（用 next 表示），有则更新组件 vnode，没有 next 指向之前的组件 vnode。为什么需要判断，这其实涉及一个组件更新策略的逻辑，我们稍后会讲。

接着是渲染新的子树 vnode，因为数据发生了变化，模板又和数据相关，所以渲染生成的子树 vnode 也会发生相应的变化。

最后就是**核心的 patch 逻辑**，用来找出新旧子树 vnode 的不同，并找到一种合适的方式更新 DOM，接下来我们就来分析这个过程。

核心逻辑：patch 流程

我们先来看 patch 流程的实现代码：

```
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 ) {

      }

  }

}

function isSameVNodeType (n1, n2) {

return n1.type === n2.type && n1.key === n2.key

}
```

在这个过程中，首先判断新旧节点是否是相同的 vnode 类型，如果不同，比如一个 div 更新成一个 ul，那么最简单的操作就是删除旧的 div 节点，再去挂载新的 ul 节点。

如果是相同的 vnode 类型，就需要走 diff 更新流程了，接着会根据不同的 vnode 类型执行不同的处理逻辑，这里我们仍然只分析普通元素类型和组件类型的处理过程。

1. 处理组件

如何**处理组件**的呢？举个例子，我们在父组件 App 中里引入了 Hello 组件：

```
<template>

<div>

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

<hello :msg="msg"></hello>

<button @click="toggle">Toggle msg</button>
```

```
</div>

</template>

<script>

export default {

    data() {

return {

msg: 'Vue'

      }

    },

methods: {

      toggle() {

this.msg = this.msg ==== 'Vue'? 'World': 'Vue'

      }

    }

  }

</script>
```

Hello 组件中是 `<div>` 包裹着一个 `<p>` 标签， 如下所示：

```
<template>

<div>
```

```
<p>Hello, {{msg}}</p>

</div>

</template>

<script>

export default {

props: {

msg: String

    }

  }

</script>
```

点击 App 组件中的按钮执行 toggle 函数，就会修改 data 中的 msg，并且会触发 App 组件的重新渲染。

结合前面对渲染函数的流程分析，这里 App 组件的根节点是 div 标签，重新渲染的子树 vnode 节点是一个普通元素的 vnode，应该先走 processElement 逻辑。组件的更新最终还是要转换成内部真实 DOM 的更新，而实际上普通元素的处理流程才是真正做 DOM 的更新，由于稍后我们会详细分析普通元素的处理流程，所以我们先跳过这里，继续往下看。

和渲染过程类似，更新过程也是一个树的深度优先遍历过程，更新完当前节点后，就会遍历更新它的子节点，因此在遍历的过程中会遇到 hello 这个组件 vnode 节点，就会执行到 processComponent 处理逻辑中，我们再来看一下它的实现，我们重点关注一下组件更新的相关逻辑：

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

if (n1 == null) {

  }

else {

    updateComponent(n1, n2, parentComponent, optimized)
```

```
  }
}
const updateComponent = (n1, n2, parentComponent, optimized) => {
const instance = (n2.component = n1.component)
if (shouldUpdateComponent(n1, n2, parentComponent, optimized)) {
    instance.next = n2
    invalidateJob(instance.update)
    instance.update()
  }
else {
    n2.component = n1.component
    n2.el = n1.el
  }
}
```

可以看到，processComponent 主要通过执行 updateComponent 函数来更新子组件，updateComponent 函数在更新子组件的时候，会先执行 shouldUpdateComponent 函数，根据新旧子组件 vnode 来判断是否需要更新子组件。这里你只需要知道，在 shouldUpdateComponent 函数的内部，主要是通过检测和对比组件 vnode 中的 props、chidren、dirs、transiton 等属性，来决定子组件是否需要更新。

这是很好理解的，因为在一个组件的子组件是否需要更新，我们主要依据子组件 vnode 是否存在一些会影响组件更新的属性变化进行判断，如果存在就会更新子组件。

虽然 Vue.js 的更新粒度是组件级别的，组件的数据变化只会影响当前组件的更新，但是在组件更新的过程中，也会对子组件做一定的检查，判断子组件是否也要更新，并通过某种机制避免子组件重复更新。

我们接着看 updateComponent 函数，如果 shouldUpdateComponent 返回 true ，那么在它的最后，先执行 invalidateJob（instance.update）避免子组件由于自身数据变化导致的重复更新，然后又执行了子组件的副作用渲染函数 instance.update 来主动触发子组件的更新。

再回到副作用渲染函数中，有了前面的讲解，我们再看组件更新的这部分代码，就能很好地理解它的逻辑了：

```
let { next, vnode } = instance

if (next) {

  updateComponentPreRender(instance, next, optimized)

}

else {

  next = vnode

}

const updateComponentPreRender = (instance, nextVNode, optimized) => {

  nextVNode.component = instance

const prevProps = instance.vnode.props

  instance.vnode = nextVNode

  instance.next = null

  updateProps(instance, nextVNode.props, prevProps, optimized)

  updateSlots(instance, nextVNode.children)

}
```

结合上面的代码，我们在更新组件的 DOM 前，需要先更新组件 vnode 节点信息，包括更改组件实例的 vnode 指针、更新 props 和更新插槽等一系列操作，因为组件在稍后执行 renderComponentRoot 时会重新渲染新的子树 vnode ，它依赖了更新后的组件 vnode 中的 props 和 slots 等数据。

所以我们现在知道了一个组件重新渲染可能会有两种场景，一种是组件本身的数据变化，这种情况下 next 是 null；另一种是父组件在更新的过程中，遇到子组件节点，先判断子组件是否需要更新，如果需要则主动执行子组件的重新渲染方法，这种情况下 next 就是新的子组件 vnode。

你可能还会有疑问，这个子组件对应的新的组件 vnode 是什么时候创建的呢？答案很简单，它是在父组件重新渲染的过程中，通过 renderComponentRoot 渲染子树 vnode 的时候生成，因为子树 vnode 是个树形结构，通过遍历它的子节点就可以访问到其对应的组件 vnode。再拿我们前面举的例子说，当 App 组件重新渲染的时候，在执行 renderComponentRoot 生成子树 vnode 的过程中，也生成了 hello 组件对应的新的组件 vnode。

所以 processComponent 处理组件 vnode，本质上就是去判断子组件是否需要更新，如果需要则递归执行子组件的副作用渲染函数来更新，否则仅仅更新一些 vnode 的属性，并让子组件实例保留对组件 vnode 的引用，用于子组件自身数据变化引起组件重新渲染的时候，在渲染函数内部可以拿到新的组件 vnode。

前面也说过，组件是抽象的，组件的更新最终还是会落到对普通 DOM 元素的更新。所以接下来我们详细分析一下组件更新中**对普通元素**的处理流程。

2. 处理普通元素

我们再来看如何处理普通元素，我把之前的示例稍加修改，将其中的 Hello 组件删掉，如下所示：

```
<template>

<div>

<p>This is {{msg}}.</p>

<button @click="toggle">Toggle msg</button>

</div>

</template>

<script>

export default {

    data() {

return {

msg: 'Vue'

      }
```

```
    },

methods: {

      toggle() {

this.msg = 'Vue'? 'World': 'Vue'

      }

    }

  }

</script>
```

当我们点击 App 组件中的按钮会执行 toggle 函数，然后修改 data 中的 msg，这就触发了 App 组件的重新渲染。

App 组件的根节点是 div 标签，重新渲染的子树 vnode 节点是一个普通元素的 vnode，所以应该先走 processElement 逻辑，我们来看这个函数的实现：

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

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

if (n1 == null) {

  }

else {

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

  }

}
```

```
const patchElement = (n1, n2, parentComponent, parentSuspense, isSVG, optimized)
=> {


const el = (n2.el = n1.el)


const oldProps = (n1 && n1.props) || EMPTY_OBJ


const newProps = n2.props || EMPTY_OBJ


  patchProps(el, n2, oldProps, newProps, parentComponent, parentSuspense, isSVG)


const areChildrenSVG = isSVG && n2.type !== 'foreignObject'


  patchChildren(n1, n2, el, null, parentComponent, parentSuspense,
areChildrenSVG)


}
```

可以看到，更新元素的过程主要做两件事情：更新 props 和更新子节点。其实这是很好理解的，因为一个 DOM 节点元素就是由它自身的一些属性和子节点构成的。

首先是更新 props，这里的 patchProps 函数就是在更新 DOM 节点的 class、style、event 以及其它的一些 DOM 属性，这个过程我不再深入分析了，感兴趣的同学可以自己看这部分代码。

其次是更新子节点，我们来看一下这里的 patchChildren 函数的实现：

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


const c1 = n1 && n1.children


const prevShapeFlag = n1 ? n1.shapeFlag : 0


const c2 = n2.children


const { shapeFlag } = n2


if (shapeFlag & 8 ) {


if (prevShapeFlag & 16 ) {


      unmountChildren(c1, parentComponent, parentSuspense)
```

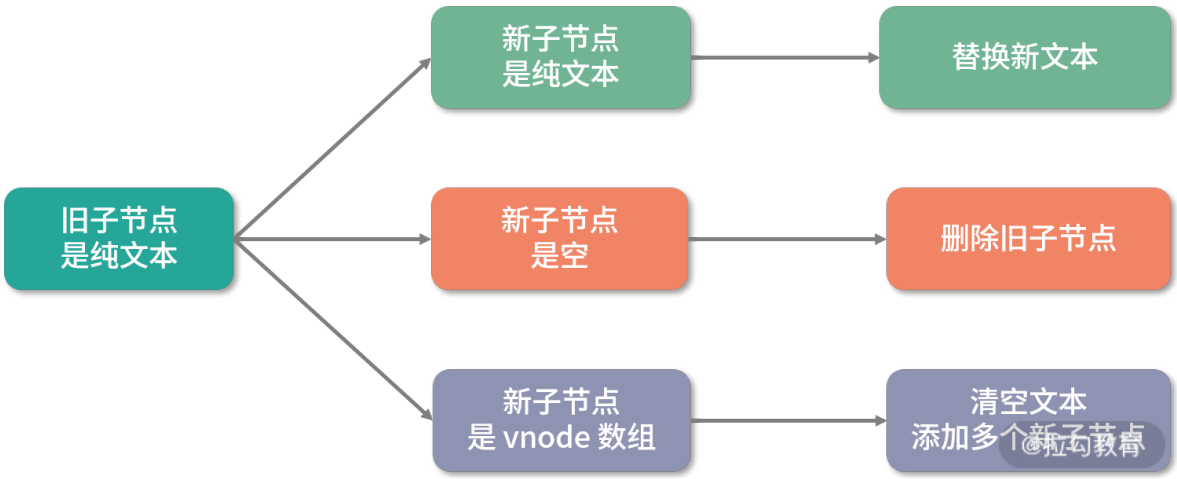
```
    }

if (c2 !== c1) {

      hostSetElementText(container, c2)

    }

  }

else {

if (prevShapeFlag & 16 ) {

if (shapeFlag & 16 ) {

        patchKeyedChildren(c1, c2, container, anchor, parentComponent,
parentSuspense, isSVG, optimized)

      }

else {

        unmountChildren(c1, parentComponent, parentSuspense, true)

      }

    }

else {

if (prevShapeFlag & 8 ) {

        hostSetElementText(container, '')

      }

if (shapeFlag & 16 ) {
```

```
        mountChildren(c2, container, anchor, parentComponent, parentSuspense,
isSVG, optimized)


      }


    }


  }


}
```

对于一个元素的子节点 vnode 可能会有三种情况：纯文本、vnode 数组和空。那么根据排列组合对于新旧子节点来说就有九种情况，我们可以通过三张图来表示。

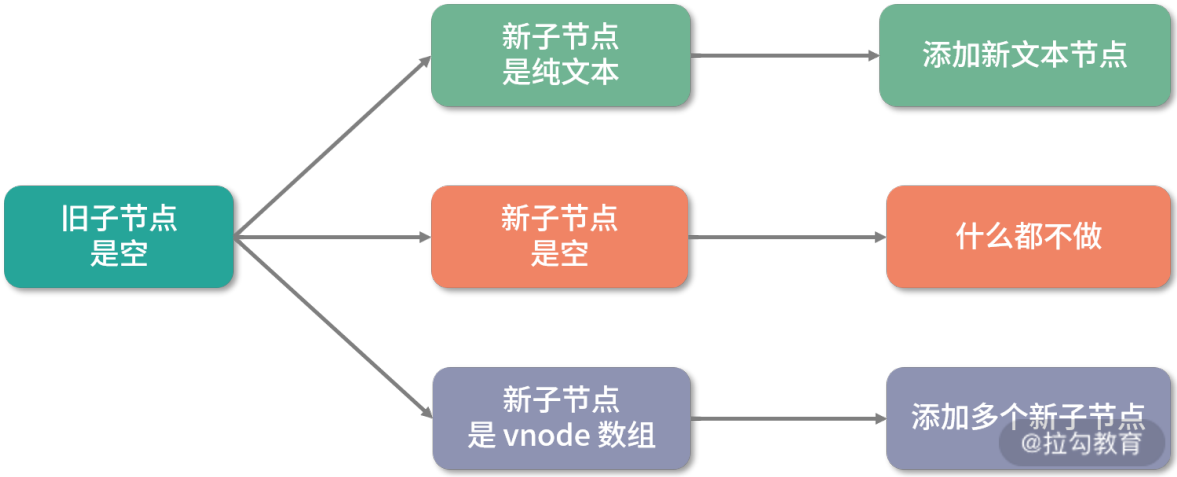
首先来看一下**旧子节点是纯文本**的情况：

- 如果新子节点也是纯文本，那么做简单地文本替换即可；
- 如果新子节点是空，那么删除旧子节点即可；
- 如果新子节点是 vnode 数组，那么先把旧子节点的文本清空，再去旧子节点的父容器下添加多个新子节点。



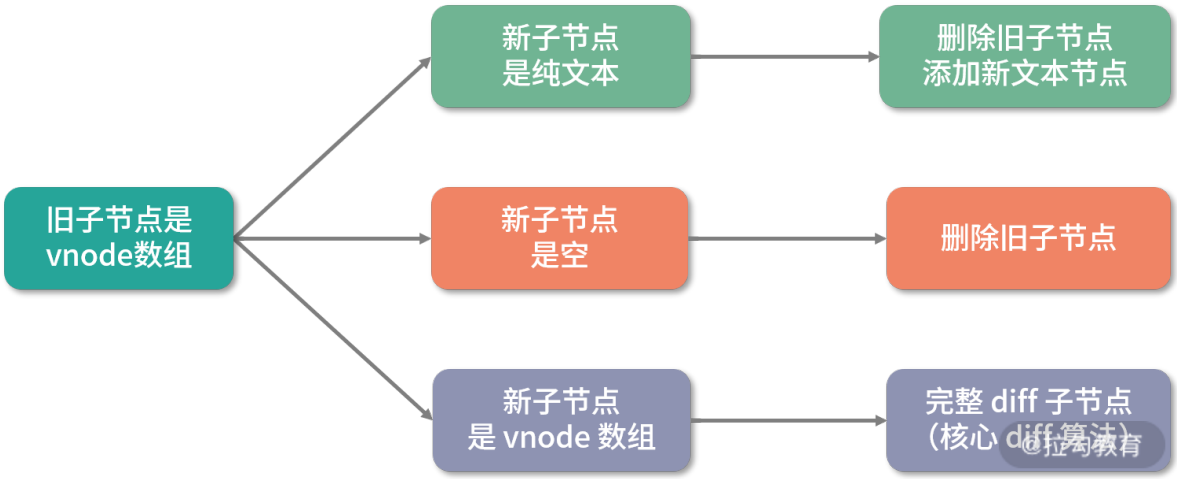
接下来看一下**旧子节点是空**的情况：

- 如果新子节点是纯文本，那么在旧子节点的父容器下添加新文本节点即可；
- 如果新子节点也是空，那么什么都不需要做；
- 如果新子节点是 vnode 数组，那么直接去旧子节点的父容器下添加多个新子节点即可。



最后来看一下**旧子节点是 vnode 数组**的情况：

- 如果新子节点是纯文本，那么先删除旧子节点，再去旧子节点的父容器下添加新文本节点；
- 如果新子节点是空，那么删除旧子节点即可；
- 如果新子节点也是 vnode 数组，那么就需要做完整的 diff 新旧子节点了，这是最复杂的情况，内部运用了核心 diff 算法。



下节课我们就来深入探究一下这个复杂的 diff 算法。

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