> 本文由 简悦 SimpRead 转码， 原文地址 kaiwu.lagou.com

通过前面的学习，我们了解到多个平行组件条件渲染，当满足条件的时候会触发某个组件的挂载，而已渲染的组件当条件不满足的时候会触发组件的卸载，举个例子：

```
<comp-a v-if="flag"></comp-a>

<comp-b v-else></comp-b>

<button @click="flag=!flag">toggle</button>
```

这里，当 flag 为 true 的时候，就会触发组件 A 的渲染，然后我们点击按钮把 flag 修改为 false，又会触发组件 A 的卸载，及组件 B 的渲染。

根据我们前面的学习，我们也知道组件的挂载和卸载都是一个递归过程，会有一定的性能损耗，对于这种可能会频繁切换的组件，我们有没有办法减少这其中的性能损耗呢？

答案是有的，Vue.js 提供了内置组件 KeepAlive，我们可以这么使用它：

```
<keep-alive>

<comp-a v-if="flag"></comp-a>

<comp-b v-else></comp-b>

<button @click="flag=!flag">toggle</button>

</keep-alive>
```

我们可以用模板导出工具看一下它编译后的 render 函数：

```
import { resolveComponent as _resolveComponent, createVNode as _createVNode,
createCommentVNode as _createCommentVNode, KeepAlive as _KeepAlive, openBlock as
_openBlock, createBlock as _createBlock } from "vue"

export function render(_ctx, _cache, $props, $setup, $data, $options) {

const _component_comp_a = _resolveComponent("comp-a")

const _component_comp_b = _resolveComponent("comp-b")

return (_openBlock(), _createBlock(_KeepAlive, null, [
```

```
    (_ctx.flag)
      ? _createVNode(_component_comp_a, { key: 0 })
      : _createVNode(_component_comp_b, { key: 1 }),
    _createVNode("button", {
      onClick: $event => (_ctx.flag=!_ctx.flag)
    }, "toggle", 8 , ["onClick"])
  ], 1024 ))
}
```

我们使用了 KeepAlive 组件对这两个组件做了一层封装，KeepAlive 是一个抽象组件，它并不会渲染成一个真实的 DOM，只会渲染内部包裹的子节点，并且让内部的子组件在切换的时候，不会走一整套递归卸载和挂载 DOM 的流程，从而优化了性能。

那么它具体是怎么做的呢？我们再来看 KeepAlive 组件的定义：

```
const KeepAliveImpl = {
  name: `KeepAlive`,
  __isKeepAlive: true,
  inheritRef: true,
  props: {
    include: [String, RegExp, Array],
    exclude: [String, RegExp, Array],
    max: [String, Number]
  },
```

```
  setup(props, { slots }) {

const cache = new Map()

const keys = new Set()

    let current = null

const instance = getCurrentInstance()

const parentSuspense = instance.suspense

const sharedContext = instance.ctx

const { renderer: { p: patch, m: move, um: _unmount, o: { createElement } } } =
sharedContext

const storageContainer = createElement('div')

    sharedContext.activate = (vnode, container, anchor, isSVG, optimized) => {

const instance = vnode.component

move(vnode, container, anchor, 0 , parentSuspense)

patch(instance.vnode, vnode, container, anchor, instance, parentSuspense, isSVG,
optimized)

queuePostRenderEffect(() => {

        instance.isDeactivated = false

if (instance.a) {

          invokeArrayFns(instance.a)

        }
```

```
const vnodeHook = vnode.props && vnode.props.onVnodeMounted

if (vnodeHook) {

          invokeVNodeHook(vnodeHook, instance.parent, vnode)

        }

      }, parentSuspense)

    }

    sharedContext.deactivate = (vnode) => {

const instance = vnode.component

move(vnode, storageContainer, null, 1 , parentSuspense)

queuePostRenderEffect(() => {

if (instance.da) {

          invokeArrayFns(instance.da)

        }

const vnodeHook = vnode.props && vnode.props.onVnodeUnmounted

if (vnodeHook) {

          invokeVNodeHook(vnodeHook, instance.parent, vnode)

        }

        instance.isDeactivated = true

      }, parentSuspense)

    }
```

```
function unmount(vnode) {
      resetShapeFlag(vnode)
      _unmount(vnode, instance, parentSuspense)
    }
function pruneCache(filter) {
      cache.forEach((vnode, key) => {
const name = getName(vnode.type)
if (name && (!filter || !filter(name))) {
          pruneCacheEntry(key)
        }
      })
    }
function pruneCacheEntry(key) {
const cached = cache.get(key)
if (!current || cached.type !== current.type) {
        unmount(cached)
      }
else if (current) {
        resetShapeFlag(current)
```

```
      }

      cache.delete(key)

      keys.delete(key)

    }

    watch(() => [props.include, props.exclude], ([include, exclude]) => {

      include && pruneCache(name => matches(include, name))

      exclude && !pruneCache(name => matches(exclude, name))

    })

    let pendingCacheKey = null

const cacheSubtree = () => {

if (pendingCacheKey != null) {

        cache.set(pendingCacheKey, instance.subTree)

      }

    }

    onBeforeMount(cacheSubtree)

    onBeforeUpdate(cacheSubtree)

    onBeforeUnmount(() => {

      cache.forEach(cached => {

const { subTree, suspense } = instance
```

```
if (cached.type === subTree.type) {

          resetShapeFlag(subTree)

const da = subTree.component.da

          da && queuePostRenderEffect(da, suspense)

return

        }

        unmount(cached)

      })

    })

return () => {

      pendingCacheKey = null

if (!slots.default) {

return null

      }

const children = slots.default()

      let vnode = children[0]

if (children.length > 1) {

if ((process.env.NODE_ENV !== 'production')) {

          warn(`KeepAlive should contain exactly one component child.`)

        }
```

```
        current = null
return children
      }
else if (!isVNode(vnode) ||
        !(vnode.shapeFlag & 4 )) {
        current = null
return vnode
      }
const comp = vnode.type
const name = getName(comp)
const { include, exclude, max } = props
if ((include && (!name || !matches(include, name))) ||
(exclude && name && matches(exclude, name))) {
return (current = vnode)
      }
const key = vnode.key == null ? comp : vnode.key
const cachedVNode = cache.get(key)
if (vnode.el) {
        vnode = cloneVNode(vnode)
```

```
      }

      pendingCacheKey = key

if (cachedVNode) {

        vnode.el = cachedVNode.el

        vnode.component = cachedVNode.component

        vnode.shapeFlag |= 512

        keys.delete(key)

        keys.add(key)

      }

else {

        keys.add(key)

if (max && keys.size > parseInt(max, 10)) {

          pruneCacheEntry(keys.values().next().value)

        }

      }

      vnode.shapeFlag |= 256

      current = vnode

return vnode

    }
```

```
  }

}
```

我把 KeepAlive 的实现拆成四个部分：**组件的渲染**、**缓存的设计**、**Props 设计**和**组件的卸载**。接下来，我们就来依次分析它们的实现。分析的过程中，我会结合前面的示例讲解，希望你也能够运行这个示例，并加入一些断点调试。

组件的渲染

首先，我们来看组件的渲染部分，可以看到 KeepAlive 组件使用了 Composition API 的方式去实现，我们已经学习过了，当 setup 函数返回的是一个函数，那么这个函数就是组件的渲染函数，我们来看它的实现：

```
return () => {

  pendingCacheKey = null

if (!slots.default) {

return null

  }

const children = slots.default()

  let vnode = children[0]

if (children.length > 1) {

if ((process.env.NODE_ENV !== 'production')) {

      warn(`KeepAlive should contain exactly one component child.`)

    }

    current = null

return children

  }
```

```
else if (!isVNode(vnode) ||

    !(vnode.shapeFlag & 4 )) {

    current = null

return vnode

  }

const comp = vnode.type

const name = getName(comp)

const { include, exclude, max } = props

if ((include && (!name || !matches(include, name))) ||

(exclude && name && matches(exclude, name))) {

return (current = vnode)

  }

const key = vnode.key == null ? comp : vnode.key

const cachedVNode = cache.get(key)

if (vnode.el) {

    vnode = cloneVNode(vnode)

  }

  pendingCacheKey = key

if (cachedVNode) {
```

```
    vnode.el = cachedVNode.el

    vnode.component = cachedVNode.component

    vnode.shapeFlag |= 512

    keys.delete(key)

    keys.add(key)

  }

else {

    keys.add(key)

if (max && keys.size > parseInt(max, 10)) {

      pruneCacheEntry(keys.values().next().value)

    }

  }

  vnode.shapeFlag |= 256

  current = vnode

return vnode

}
```

函数先从 slots.default() 拿到子节点 children，它就是 KeepAlive 组件包裹的子组件，由于 KeepAlive 只能渲染单个子节点，所以当 children 长度大于 1 的时候会报警告。

我们先不考虑缓存部分，KeepAlive 渲染的 vnode 就是子节点 children 的第一个元素，它是函数的返回值。

因此我们说 KeepAlive 是抽象组件，它本身不渲染成实体节点，而是渲染它的第一个子节点。

当然，没有缓存的 KeepAlive 组件是没有灵魂的，这种抽象的封装也是没有任何意义的，所以接下来我们重点来看它的缓存是如何设计的。

缓存的设计

我们先来思考一件事情，我们需要缓存什么？

组件的递归 patch 过程，主要就是为了渲染 DOM，显然这个递归过程是有一定的性能耗时的，既然目标是为了渲染 DOM，那么我们是不是可以把 DOM 缓存了，这样下一次渲染我们就可以直接从缓存里获取 DOM 并渲染，就不需要每次都重新递归渲染了。

实际上 KeepAlive 组件就是这么做的，它注入了两个钩子函数，onBeforeMount 和 onBeforeUpdate，在这两个钩子函数内部都执行了 cacheSubtree 函数来做缓存：

```
const cacheSubtree = () => {

if (pendingCacheKey != null) {

    cache.set(pendingCacheKey, instance.subTree)

  }

}
```

由于 pendingCacheKey 是在 KeepAlive 的 render 函数中才会被赋值，所以 KeepAlive 首次进入 onBeforeMount 钩子函数的时候是不会缓存的。

然后 KeepAlive 执行 render 的时候，pendingCacheKey 会被赋值为 vnode.key，我们回过头看一下示例渲染后的模板：

```
import { resolveComponent as _resolveComponent, createVNode as _createVNode,
createCommentVNode as _createCommentVNode, KeepAlive as _KeepAlive, openBlock as
_openBlock, createBlock as _createBlock } from "vue"

export function render(_ctx, _cache, $props, $setup, $data, $options) {

const _component_comp_a = _resolveComponent("comp-a")

const _component_comp_b = _resolveComponent("comp-b")

return (_openBlock(), _createBlock(_KeepAlive, null, [

    (_ctx.flag)

      ? _createVNode(_component_comp_a, { key: 0 })
```

```
      : _createVNode(_component_comp_b, { key: 1 }),

    _createVNode("button", {

      onClick: $event => (_ctx.flag=!_ctx.flag)

    }, "toggle", 8 , ["onClick"])

  ], 1024 ))

}
```

我们注意到 KeepAlive 的子节点创建的时候都添加了一个 key 的 prop，它就是专门为 KeepAlive 的缓存设计的，这样每一个子节点都能有一个唯一的 key。

页面首先渲染 A 组件，接着当我们点击按钮的时候，修改了 flag 的值，会触发当前组件的重新渲染，进而也触发了 KeepAlvie 组件的重新渲染，在组件重新渲染前，会执行 onBeforeUpdate 对应的钩子函数，也就再次执行到 cacheSubtree 函数中。

这个时候 pendingCacheKey 对应的是 A 组件 vnode 的 key，instance.subTree 对应的也是 A 组件的渲染子树，所以 KeepAlive 每次在更新前，会缓存前一个组件的渲染子树。

> 经过前面的分析，我认为 onBeforeMount 的钩子函数注入似乎并没有必要，我在源码中删除后再跑 Vue.js 3.0 的单测也能通过，如果你有不同意见，欢迎在留言区与我分享。

这个时候渲染了 B 组件，当我们再次点击按钮，修改 flag 值的时候，会再次触发 KeepAlvie 组件的重新渲染，当然此时执行 onBeforeUpdate 钩子函数缓存的就是 B 组件的渲染子树了。

接着再次执行 KeepAlive 组件的 render 函数，此时就可以从缓存中根据 A 组件的 key 拿到对应的渲染子树 cachedVNode 的了，然后执行如下逻辑：

```
if (cachedVNode) {

  vnode.el = cachedVNode.el

  vnode.component = cachedVNode.component

  vnode.shapeFlag |= 512

  keys.delete(key)

  keys.add(key)

}
```

```
else {

  keys.add(key)

if (max && keys.size > parseInt(max, 10)) {

    pruneCacheEntry(keys.values().next().value)

  }

}
```

有了缓存的渲染子树后，我们就可以直接拿到它对应的 DOM 以及组件实例 component，赋值给 KeepAlive 的 vnode，并更新 vnode.shapeFlag，以便后续 patch 阶段使用。

> 注意，这里有一个额外的缓存管理的逻辑，我们稍后讲 Props 设计的时候会详细说。

那么，对于 KeepAlive 组件的渲染来说，有缓存和没缓存在 patch 阶段有何区别呢，由于 KeepAlive 缓存的都是有状态的组件 vnode，我们再来回顾一下 patchComponent 函数的实现：

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

if (n1 == null) {

if (n2.shapeFlag & 512 ) {

      parentComponent.ctx.activate(n2, container, anchor, isSVG, optimized)

    }

else {

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

    }

  }

else {
```

```
  }

}
```

KeepAlive 首次渲染某一个子节点时，和正常的组件节点渲染没有区别，但是有缓存后，由于标记了 shapeFlag，所以在执行 processComponent 函数时会走到处理 KeepAlive 组件的逻辑中，执行 KeepAlive 组件实例上下文中的 activate 函数，我们来看它的实现：

```
sharedContext.activate = (vnode, container, anchor, isSVG, optimized) => {

const instance = vnode.component

move(vnode, container, anchor, 0 , parentSuspense)

patch(instance.vnode, vnode, container, anchor, instance, parentSuspense, isSVG,
optimized)

queuePostRenderEffect(() => {

    instance.isDeactivated = false

if (instance.a) {

      invokeArrayFns(instance.a)

    }

const vnodeHook = vnode.props && vnode.props.onVnodeMounted

if (vnodeHook) {

      invokeVNodeHook(vnodeHook, instance.parent, vnode)

    }

  }, parentSuspense)

}
```

可以看到，由于此时已经能从 vnode.el 中拿到缓存的 DOM 了，所以可以直接调用 move 方法挂载节点，然后执行 patch 方法更新组件，以防止 props 发生变化的情况。

接下来，就是通过 queuePostRenderEffect 的方式，在组件渲染完毕后，执行子节点组件定义的 activated 钩子函数。

至此，我们就了解了 KeepAlive 的缓存设计，KeepAlive 包裹的子组件在其渲染后，下一次 KeepAlive 组件更新前会被缓存，缓存后的子组件在下一次渲染的时候直接从缓存中拿到子树 vnode 以及对应的 DOM 元素，直接渲染即可。

当然，光有缓存还不够灵活，有些时候我们想针对某些子组件缓存，某些子组件不缓存，另外，我们还想限制 KeepAlive 组件的最大缓存个数，怎么办呢？KeepAlive 设计了几个 Props，允许我们可以对上述需求做配置。

Props 设计

KeepAlive 一共支持了三个 Props，分别是 include、exclude 和 max。

```
props: {

include: [String, RegExp, Array],

exclude: [String, RegExp, Array],

max: [String, Number]

}
```

include 和 exclude 对应的实现逻辑如下：

```
const { include, exclude, max } = props

if ((include && (!name || !matches(include, name))) ||

(exclude && name && matches(exclude, name))) {

return (current = vnode)

}
```

很好理解，如果子组件名称不匹配 include 的 vnode ，以及子组件名称匹配 exclude 的 vnode 都不应该被缓存，而应该直接返回。

当然，由于 props 是响应式的，在 include 和 exclude props 发生变化的时候也应该有相关的处理逻辑，如下：

```
watch(() => [props.include, props.exclude], ([include, exclude]) => {

  include && pruneCache(name => matches(include, name))

  exclude && !pruneCache(name => matches(exclude, name))

})
```

监听的逻辑也很简单，当 include 发生变化的时候，从缓存中删除那些 name 不匹配 include 的 vnode 节点；当 exclude 发生变化的时候，从缓存中删除那些 name 匹配 exclude 的 vnode 节点。

除了 include 和 exclude 之外，KeepAlive 组件还支持了 max prop 来控制缓存的最大个数。

由于缓存本身就是占用了内存，所以有些场景我们希望限制 KeepAlive 缓存的个数，这时我们可以通过 max 属性来控制，当缓存新的 vnode 的时候，会做一定程度的缓存管理，如下：

```
keys.add(key)

if (max && keys.size > parseInt(max, 10)) {

pruneCacheEntry(keys.values().next().value)

}
```

由于新的缓存 key 都是在 keys 的结尾添加的，所以当缓存的个数超过 max 的时候，就从最前面开始删除，符合 LRU 最近最少使用的算法思想。

组件的卸载

了解完 KeepAlive 组件的渲染、缓存和 Props 设计后，我们接着来看 KeepAlive 组件的卸载过程。

我们先来分析 KeepAlive 内部包裹的子组件的卸载过程，前面我们提到 KeepAlive 渲染的过程实际上是渲染它的第一个子组件节点，并且会给渲染的 vnode 打上如下标记：

加上这个 shapeFlag 有什么用呢，我们结合前面的示例来分析。

```
<keep-alive>

<comp-a v-if="flag"></comp-a>

<comp-b v-else></comp-b>

<button @click="flag=!flag">toggle</button>

</keep-alive>
```

当 flag 为 true 的时候，渲染 A 组件，然后我们点击按钮修改 flag 的值，会触发 KeepAlive 组件的重新渲染，会先执行 BeforeUpdate 钩子函数缓存 A 组件对应的渲染子树 vnode，然后再执行 patch 更新子组件。

这个时候会执行 B 组件的渲染，以及 A 组件的卸载，我们知道组件的卸载会执行 unmount 方法，其中有一个关于 KeepAlive 组件的逻辑，如下：

```
const unmount = (vnode, parentComponent, parentSuspense, doRemove = false) => {

const { shapeFlag  } = vnode

if (shapeFlag & 256 ) {

    parentComponent.ctx.deactivate(vnode)

return

  }

}
```

如果 shapeFlag 满足 KeepAlive 的条件，则执行相应的 deactivate 函数，它的定义如下：

```
sharedContext.deactivate = (vnode) => {

const instance = vnode.component

move(vnode, storageContainer, null, 1 , parentSuspense)

queuePostRenderEffect(() => {

if (instance.da) {

      invokeArrayFns(instance.da)

    }

const vnodeHook = vnode.props && vnode.props.onVnodeUnmounted
```

```
if (vnodeHook) {
      invokeVNodeHook(vnodeHook, instance.parent, vnode)
    }
    instance.isDeactivated = true
  }, parentSuspense)
}
```

函数首先通过 move 方法从 DOM 树中移除该节点，接着通过 queuePostRenderEffect 的方式执行定义的 deactivated 钩子函数。

注意，这里我们只是移除了 DOM，并没有真正意义上的执行子组件的整套卸载流程。

那么除了点击按钮引起子组件的卸载之外，当 KeepAlive 所在的组件卸载时，由于卸载的递归特性，也会触发 KeepAlive 组件的卸载，在卸载的过程中会执行 onBeforeUnmount 钩子函数，如下：

```
onBeforeUnmount(() => {
  cache.forEach(cached => {
const { subTree, suspense } = instance
if (cached.type === subTree.type) {
      resetShapeFlag(subTree)
const da = subTree.component.da
      da && queuePostRenderEffect(da, suspense)
return
    }
    unmount(cached)
  })
```

```
})
```

它会遍历所有缓存的 vnode，并且比对缓存的 vnode 是不是当前 KeepAlive 组件渲染的 vnode。

如果是的话，则执行 resetShapeFlag 方法，它的作用是修改 vnode 的 shapeFlag，不让它再被当作一个 KeepAlive 的 vnode 了，这样就可以走正常的卸载逻辑。接着通过 queuePostRenderEffect 的方式执行子组件的 deactivated 钩子函数。

如果不是，则执行 unmount 方法重置 shapeFlag 以及执行缓存 vnode 的整套卸载流程。

总结

好的，到这里我们这一节的学习也要结束啦，通过这节课的学习，你应该明白 KeepAlive 实际上是一个抽象节点，渲染的是它的第一个子节点，并了解它的缓存设计、Props 设计和卸载过程。

最后，给你留一道思考题，我们是如何给组件注册 activated 和 deactivated 钩子函数的，它们的执行和其他钩子函数比，有什么不同？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-core/src/components/KeepAlive.ts
> packages/runtime-core/src/renderer.ts