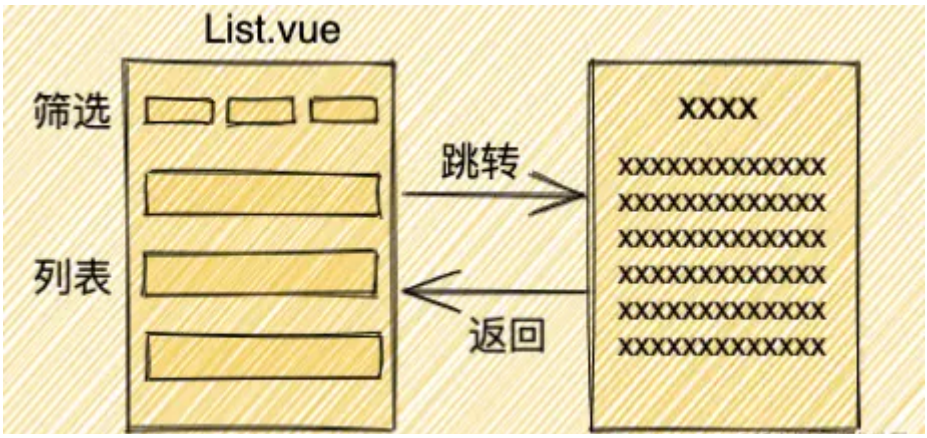
「Vue源码学习」简单讲一讲keep-alive的原理吧

今天，就给大家讲讲Vue中常用的组件 `keep-alive` 的基本原理吧！

场景

可能大家在平时的开发中会经常遇到这样的场景：有一个可以进行筛选的列表页 `List.vue` ，点击某一项时进入相应的详情页面，等到你从详情页返回 `List.vue` 时，发现列表页居然刷新了！刚刚的筛选条件都没了！！！



keep-alive

是什么?

- `keep-alive` 是一个 `Vue全局组件`
- `keep-alive` 本身不会渲染出来，也不会出现在父组件链中
- `keep-alive` 包裹动态组件时，会缓存不活动的组件，而不是销毁它们

怎么用?

`keep-alive` 接收三个参数：

- `include` ：可传 `字符串、正则表达式、数组` ，名称匹配成功的组件会被缓存
- `exclude` ：可传 `字符串、正则表达式、数组` ，名称匹配成功的组件不会被缓存
- `max` ：可传 `数字` ，限制缓存组件的最大数量

`include` 和 `exclude` ，传 `数组` 情况居多

动态组件

```vue
<keep-alive :include="allowList" :exclude="noAllowList" :max="amount">
    <component :is="currentComponent"></component>
</keep-alive>
```

路由组件

```vue
<keep-alive :include="allowList" :exclude="noAllowList" :max="amount">
    <router-view></router-view>
</keep-alive>
```

源码

组件基础

前面说了，`keep-alive`是一个`Vue全局组件`，他接收三个参数：

- `include`：可传`字符串、正则表达式、数组`，名称匹配成功的组件会被缓存
- `exclude`：可传`字符串、正则表达式、数组`，名称匹配成功的组件不会被缓存
- `max`：可传`数字`，限制缓存组件的最大数量，超过`max`则按照`LRU算法`进行置换

顺便说说`keep-alive`在各个生命周期里都做了啥吧：

- `created`：初始化一个`cache、keys`，前者用来存缓存组件的虚拟dom集合，后者用来存缓存组件的key集合
- `mounted`：实时监听`include、exclude`这两个的变化，并执行相应操作
- `destroyed`：删除掉所有缓存相关的东西

> 之前说了，`keep-alive`不会被渲染到页面上，所以`abstract`这个属性至关重要！

```vue
// src/core/components/keep-alive.js

export default {
  name: 'keep-alive',
  abstract: true, // 判断此组件是否需要在渲染成真实DOM
  props: {
    include: patternTypes,
    exclude: patternTypes,
    max: [String, Number]
  },
  created() {
    this.cache = Object.create(null) // 创建对象来存储  缓存虚拟dom
```

```
    this.keys = [] // 创建数组来存储  缓存key
  },
  mounted() {
    // 实时监听include、exclude的变动
    this.$watch('include', val => {
      pruneCache(this, name => matches(val, name))
    })
    this.$watch('exclude', val => {
      pruneCache(this, name => !matches(val, name))
    })
  },
  destroyed() {
    for (const key in this.cache) { // 删除所有的缓存
      pruneCacheEntry(this.cache, key, this.keys)
    }
  },
  render() {
      // 下面讲
  }
}

```

pruneCacheEntry函数

咱们上面实现的生命周期 `destroyed` 中，执行了 `删除所有缓存` 这个操作，而这个操作是通过调用 `pruneCacheEntry` 来实现的，那咱们来说说 `pruneCacheEntry` 里做了啥吧

```
// src/core/components/keep-alive.js

function pruneCacheEntry (
  cache: VNodeCache,
  key: string,
  keys: Array<string>,
  current?: VNode
) {
  const cached = cache[key]
  if (cached && (!current || cached.tag !== current.tag)) {
    cached.componentInstance.$destroy() // 执行组件的destory钩子函数
  }
  cache[key] = null  // 设为null
  remove(keys, key) // 删除对应的元素
}
```

总结一下就是做了三件事：

- 1、遍历集合，执行所有缓存组件的 `$destroy` 方法
- 2、将 `cache` 对应 `key` 的内容设置为 `null`
- 3、删除 `keys` 中对应的元素

render函数

> 以下称 `include` 为白名单， `exclude` 为黑名单 `render` 函数里主要做了这些事：

- 第一步：获取到 `keep-alive` 包裹的第一个组件以及它的 `组件名称`
- 第二步：判断此 `组件名称` 是否能被 `白名单、黑名单` 匹配，如果 `不能被白名单匹配 || 能被黑名单匹配` ，则直接返回 `VNode` ，不往下执行，如果不符合，则往下执行 `第三步`
- 第三步：根据 `组件ID、tag` 生成 `缓存key` ，并在缓存集合中查找是否已缓存过此组件。如果已缓存过，直接取出缓存组件，并更新 `缓存key` 在 `keys` 中的位置（这是 `LRU算法` 的关键），如果没缓存过，则继续 `第四步`
- 第四步：分别在 `cache、keys` 中保存 `此组件` 以及他的 `缓存key` ，并检查数量是否超过 `max` ，超过则根据 `LRU算法` 进行删除
- 第五步：将此组件实例的 `keepAlive` 属性设置为true，这很重要哦，下面会讲到的！

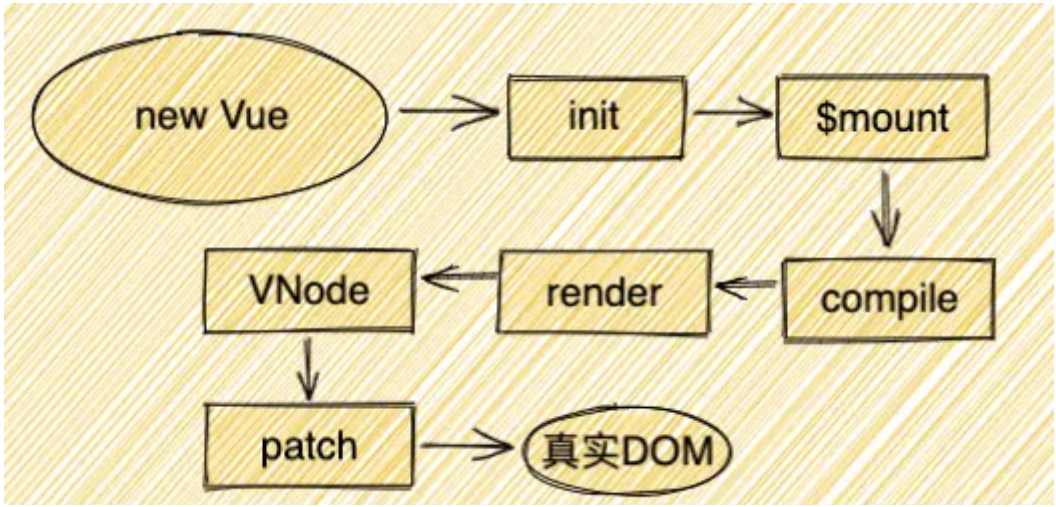
```
// src/core/components/keep-alive.js

render() {
  const slot = this.$slots.default
  const vnode: VNode = getFirstComponentChild(slot) // 找到第一个子组件对象
  const componentOptions: ?VNodeComponentOptions = vnode &&
vnode.componentOptions
  if (componentOptions) { // 存在组件参数
    // check pattern
    const name: ?string = getComponentName(componentOptions) // 组件名
    const { include, exclude } = this
    if ( // 条件匹配
      // not included
      (include && (!name || !matches(include, name))) ||
      // excluded
      (exclude && name && matches(exclude, name))
    ) {
      return vnode
    }

    const { cache, keys } = this
    const key: ?string = vnode.key == null // 定义组件的缓存key
      // same constructor may get registered as different local components
      // so cid alone is not enough (#3269)
```

```
24        ? componentOptions.Ctor.cid + (componentOptions.tag ?
`::${componentOptions.tag}` : '')
25        : vnode.key
26      if (cache[key]) { // 已经缓存过该组件
27        vnode.componentInstance = cache[key].componentInstance
28        // make current key freshest
29        remove(keys, key)
30        keys.push(key) // 调整key排序
31      } else {
32        cache[key] = vnode // 缓存组件对象
33        keys.push(key)
34        // prune oldest entry
35        if (this.max && keys.length > parseInt(this.max)) { // 超过缓存数限制，将第
一个删除
36          pruneCacheEntry(cache, keys[0], keys, this._vnode)
37        }
38      }
39
40      vnode.data.keepAlive = true // 渲染和执行被包裹组件的钩子函数需要用到
41    }
42    return vnode || (slot && slot[0])
```

渲染

咱们先来看看Vue一个组件是怎么渲染的，咱们从 `render` 开始说：

- `render`：此函数会将组件转成 `VNode`
- `patch`：此函数在初次渲染时会直接渲染根据拿到的 `VNode` 直接渲染成 `真实DOM`，第二次渲染开始就会拿 `VNode` 会跟 `旧VNode` 对比，打补丁（diff算法对比发生在此阶段），然后渲染成 `真实DOM`



keep-alive本身渲染

刚刚说了，`keep-alive`自身组件不会被渲染到页面上，那是怎么做到的呢？其实就是通过判断组件实例上的`abstract`的属性值，如果是`true`的话，就跳过该实例，该实例也不会出现在父级链上

```
// src/core/instance/lifecycle.js

export function initLifecycle (vm: Component) {
  const options = vm.$options
  // 找到第一个非abstract的父组件实例
  let parent = options.parent
  if (parent && !options.abstract) {
    while (parent.$options.abstract && parent.$parent) {
      parent = parent.$parent
    }
    parent.$children.push(vm)
  }
  vm.$parent = parent
  // ...
}
```

包裹组件渲染

咱们再来说说被`keep-alive`包裹着的组件是如何使用缓存的吧。刚刚说了`VNode -> 真实DOM`是发生在`patch`的阶段，而其实这也是要细分的：`VNode -> 实例化 -> _update -> 真实DOM`，而组件使用缓存的判断就发生在`实例化`这个阶段，而这个阶段调用的是`createComponent`函数，那我们就来说说这个函数吧：

```
// src/core/vdom/patch.js

function createComponent (vnode, insertedVnodeQueue, parentElm, refElm) {
  let i = vnode.data
  if (isDef(i)) {
    const isReactivated = isDef(vnode.componentInstance) && i.keepAlive
    if (isDef(i = i.hook) && isDef(i = i.init)) {
      i(vnode, false /* hydrating */)
    }

    if (isDef(vnode.componentInstance)) {
      initComponent(vnode, insertedVnodeQueue)
      insert(parentElm, vnode.elm, refElm) // 将缓存的DOM（vnode.elm）插入父元素中
      if (isTrue(isReactivated)) {
        reactivateComponent(vnode, insertedVnodeQueue, parentElm, refElm)
      }
      return true
```

```
    }
  }
}
```

- 在第一次加载被包裹组件时，因为 `keep-alive` 的 `render` 先于包裹组件加载之前执行，所以此时 `vnode.componentInstance` 的值是 `undefined` ，而 `keepAlive` 是 `true` ，则代码走到 `i(vnode, false /* hydrating */)` 就不往下走了
- 再次访问包裹组件时， `vnode.componentInstance` 的值就是已经缓存的组件实例，那么会执行 `insert(parentElm, vnode.elm, refElm)` 逻辑，这样就直接把上一次的DOM插入到了父元素中。