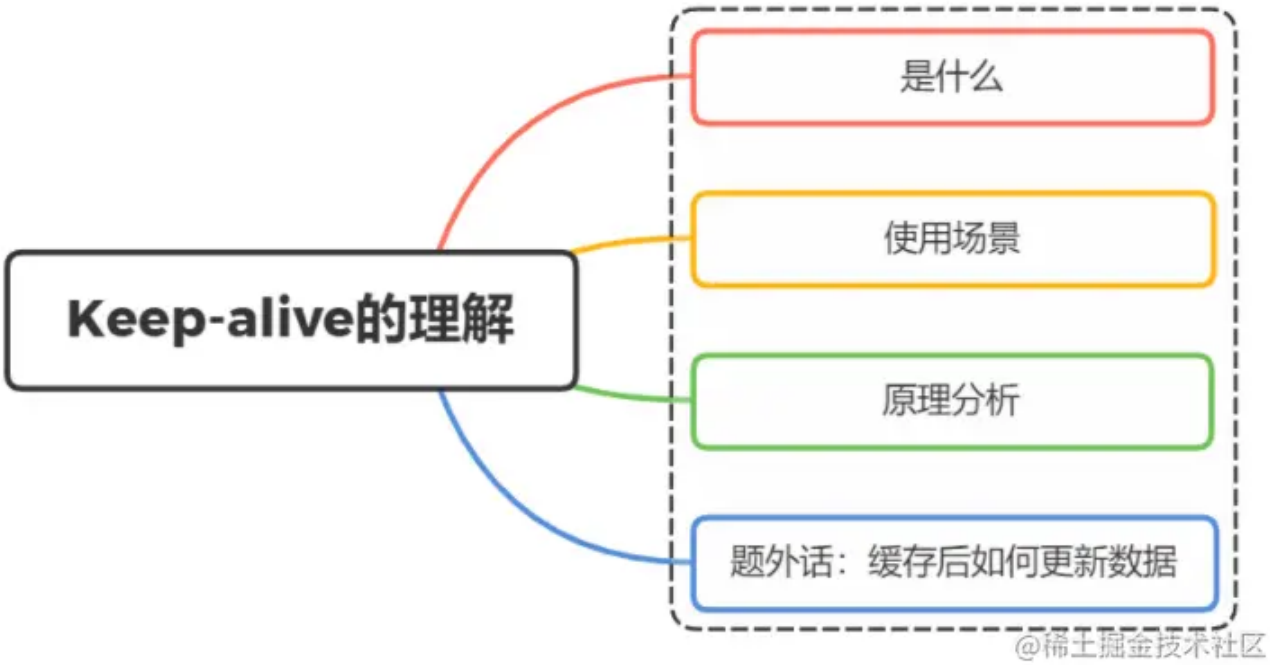
说说你对keep-alive的理解是什么?



一、Keep-alive 是什么

`keep-alive` 是 `vue` 中的内置组件，能在组件切换过程中将状态保留在内存中，防止重复渲染 `DOM`

`keep-alive` 包裹动态组件时，会缓存不活动的组件实例，而不是销毁它们

`keep-alive` 可以设置以下 `props` 属性：

- `include` - 字符串或正则表达式。只有名称匹配的组件会被缓存
- `exclude` - 字符串或正则表达式。任何名称匹配的组件都不会被缓存
- `max` - 数字。最多可以缓存多少组件实例

关于 `keep-alive` 的基本用法：

```ruby
<keep-alive>
  <component :is="view"></component>
</keep-alive>
```

使用 `includes` 和 `exclude`：

```xml
<keep-alive include="a,b">
  <component :is="view"></component>
</keep-alive>

<!-- 正则表达式 (使用 `v-bind`) -->
<keep-alive :include="/a|b/">
  <component :is="view"></component>
</keep-alive>

<!-- 数组 (使用 `v-bind`) -->
<keep-alive :include="['a', 'b']">
  <component :is="view"></component>
</keep-alive>
```

匹配首先检查组件自身的 `name` 选项，如果 `name` 选项不可用，则匹配它的局部注册名称 (父组件 `components` 选项的键值)，匿名组件不能被匹配

设置了 keep-alive 缓存的组件，会多出两个生命周期钩子（`activated` 与 `deactivated`）：

- 首次进入组件时：
  `beforeRouteEnter` > `beforeCreate` > `created` > `mounted` > `activated` > … …
  > `beforeRouteLeave` > `deactivated`
- 再次进入组件时：`beforeRouteEnter` > `activated` > … …
  > `beforeRouteLeave` > `deactivated`

二、使用场景

使用原则：当我们在某些场景下不需要让页面重新加载时我们可以使用 `keepalive`

举个栗子:

当我们从 `首页` –> `列表页` –> `商详页` –> `再返回`，这时候列表页应该是需要 `keep-alive`

从 `首页` –> `列表页` –> `商详页` –> `返回到列表页(需要缓存)` –> `返回到首页(需要缓存)` –> `再次进入列表页(不需要缓存)`，这时候可以按需来控制页面的 `keep-alive`

在路由中设置 `keepAlive` 属性判断是否需要缓存

```css
{
  path: 'list',
  name: 'itemList', // 列表页
  component (resolve) {
    require(['@/pages/item/list'], resolve)
 },
 meta: {
  keepAlive: true,
  title: '列表页'
 }
}
```

使用 `<keep-alive>`

```xml
<div id="app" class='wrapper'>
    <keep-alive>
        <!-- 需要缓存的视图组件 -->
        <router-view v-if="$route.meta.keepAlive"></router-view>
     </keep-alive>
      <!-- 不需要缓存的视图组件 -->
     <router-view v-if="!$route.meta.keepAlive"></router-view>
</div>
```

三、原理分析

`keep-alive` 是 `vue` 中内置的一个组件

源码位置：src/core/components/keep-alive.js

```kotlin
export default {
  name: 'keep-alive',
  abstract: true,

  props: {
    include: [String, RegExp, Array],
    exclude: [String, RegExp, Array],
    max: [String, Number]
  },

  created () {
    this.cache = Object.create(null)
    this.keys = []
  },
```

```
destroyed () {
  for (const key in this.cache) {
    pruneCacheEntry(this.cache, key, this.keys)
  }
},

mounted () {
  this.$watch('include', val => {
    pruneCache(this, name => matches(val, name))
  })
  this.$watch('exclude', val => {
    pruneCache(this, name => !matches(val, name))
  })
},

render() {
  /* 获取默认插槽中的第一个组件节点 */
  const slot = this.$slots.default
  const vnode = getFirstComponentChild(slot)
  /* 获取该组件节点的componentOptions */
  const componentOptions = vnode && vnode.componentOptions

  if (componentOptions) {
    /* 获取该组件节点的名称，优先获取组件的name字段，如果name不存在则获取组件的tag */
    const name = getComponentName(componentOptions)

    const { include, exclude } = this
    /* 如果name不在inlcude中或者存在于exlude中则表示不缓存，直接返回vnode */
    if (
      (include && (!name || !matches(include, name))) ||
      // excluded
      (exclude && name && matches(exclude, name))
    ) {
      return vnode
    }

    const { cache, keys } = this
    /* 获取组件的key值 */
    const key = vnode.key == null
      // same constructor may get registered as different local components
      // so cid alone is not enough (#3269)
      ? componentOptions.Ctor.cid + (componentOptions.tag ? `::${componentOptions.tag}` : '')
      : vnode.key
   /*  拿到key值后去this.cache对象中去寻找是否有该值，如果有则表示该组件有缓存，即命中缓存 */
    if (cache[key]) {
      vnode.componentInstance = cache[key].componentInstance
      // make current key freshest
      remove(keys, key)
```

```
        keys.push(key)
      }
        /* 如果没有命中缓存，则将其设置进缓存 */
        else {
        cache[key] = vnode
        keys.push(key)
        // prune oldest entry
        /* 如果配置了max并且缓存的长度超过了this.max，则从缓存中删除第一个 */
        if (this.max && keys.length > parseInt(this.max)) {
          pruneCacheEntry(cache, keys[0], keys, this._vnode)
        }
      }

      vnode.data.keepAlive = true
    }
    return vnode || (slot && slot[0])
  }
}
```

可以看到该组件没有 `template`，而是用了 `render`，在组件渲染的时候会自动执行 `render` 函数

`this.cache` 是一个对象，用来存储需要缓存的组件，它将以如下形式存储：

kotlin 复制代码

```
this.cache = {
    'key1':'组件1',
    'key2':'组件2',
    // ...
}
```

在组件销毁的时候执行 `pruneCacheEntry` 函数

php 复制代码

```
function pruneCacheEntry (
  cache: VNodeCache,
  key: string,
  keys: Array<string>,
  current?: VNode
) {
  const cached = cache[key]
  /* 判断当前没有处于被渲染状态的组件，将其销毁*/
  if (cached && (!current || cached.tag !== current.tag)) {
    cached.componentInstance.$destroy()
  }
  cache[key] = null
  remove(keys, key)
}
```

在 `mounted` 钩子函数中观测 `include` 和 `exclude` 的变化，如下：

```kotlin
mounted () {
    this.$watch('include', val => {
        pruneCache(this, name => matches(val, name))
    })
    this.$watch('exclude', val => {
        pruneCache(this, name => !matches(val, name))
    })
}
```

如果 `include` 或 `exclude` 发生了变化，即表示定义需要缓存的组件的规则或者不需要缓存的组件的规则发生了变化，那么就执行 `pruneCache` 函数，函数如下：

```scss
function pruneCache (keepAliveInstance, filter) {
  const { cache, keys, _vnode } = keepAliveInstance
  for (const key in cache) {
    const cachedNode = cache[key]
    if (cachedNode) {
      const name = getComponentName(cachedNode.componentOptions)
      if (name && !filter(name)) {
        pruneCacheEntry(cache, key, keys, _vnode)
      }
    }
  }
}
```

在该函数内对 `this.cache` 对象进行遍历，取出每一项的 `name` 值，用其与新的缓存规则进行匹配，如果匹配不上，则表示在新的缓存规则下该组件已经不需要被缓存，则调用 `pruneCacheEntry` 函数将其从 `this.cache` 对象剔除即可

关于 `keep-alive` 的最强大缓存功能是在 `render` 函数中实现

首先获取组件的 `key` 值：

```javascript
const key = vnode.key == null?
componentOptions.Ctor.cid + (componentOptions.tag ? `::${componentOptions.tag}` : '')
: vnode.key
```

拿到 `key` 值后去 `this.cache` 对象中去寻找是否有该值，如果有则表示该组件有缓存，即命中缓存，如下：

```scss
/* 如果命中缓存，则直接从缓存中拿 vnode 的组件实例 */
if (cache[key]) {
    vnode.componentInstance = cache[key].componentInstance
    /* 调整该组件key的顺序，将其从原来的地方删掉并重新放在最后一个 */
    remove(keys, key)
    keys.push(key)
}
```

直接从缓存中拿 `vnode` 的组件实例，此时重新调整该组件 `key` 的顺序，将其从原来的地方删掉并重新放在 `this.keys` 中最后一个

`this.cache` 对象中没有该 `key` 值的情况，如下：

```kotlin
/* 如果没有命中缓存，则将其设置进缓存 */
else {
    cache[key] = vnode
    keys.push(key)
    /* 如果配置了max并且缓存的长度超过了this.max，则从缓存中删除第一个 */
    if (this.max && keys.length > parseInt(this.max)) {
        pruneCacheEntry(cache, keys[0], keys, this._vnode)
    }
}
```

表明该组件还没有被缓存过，则以该组件的 `key` 为键，组件 `vnode` 为值，将其存入 `this.cache` 中，并且把 `key` 存入 `this.keys` 中

此时再判断 `this.keys` 中缓存组件的数量是否超过了设置的最大缓存数量值 `this.max` ，如果超过了，则把第一个缓存组件删掉

四、思考题：缓存后如何获取数据

解决方案可以有以下两种：

- beforeRouteEnter
- actived

beforeRouteEnter

每次组件渲染的时候，都会执行 `beforeRouteEnter`

```javascript 复制代码
beforeRouteEnter(to, from, next){
    next(vm=>{
        console.log(vm)
        // 每次进入路由执行
        vm.getData()  // 获取数据
    })
},
```

actived

在 `keep-alive` 缓存的组件被激活的时候，都会执行 `actived` 钩子

```javascript 复制代码
activated(){
   this.getData() // 获取数据
},
```

注意：服务器端渲染期间 `avtived` 不被调用