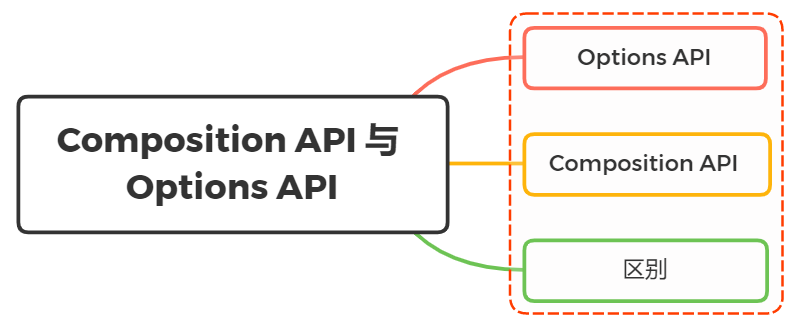
# 面试官：Vue3.0 所采用的 Composition Api 与 Vue2.x 使用的 Options Api 有什么不同？



## 开始之前

Composition API 可以说是Vue3的最大特点，那么为什么要推出Composition Api，解决了什么问题？

通常使用Vue2开发的项目，普遍会存在以下问题：

* 代码的可读性随着组件变大而变差
* 每一种代码复用的方式，都存在缺点
* TypeScript支持有限

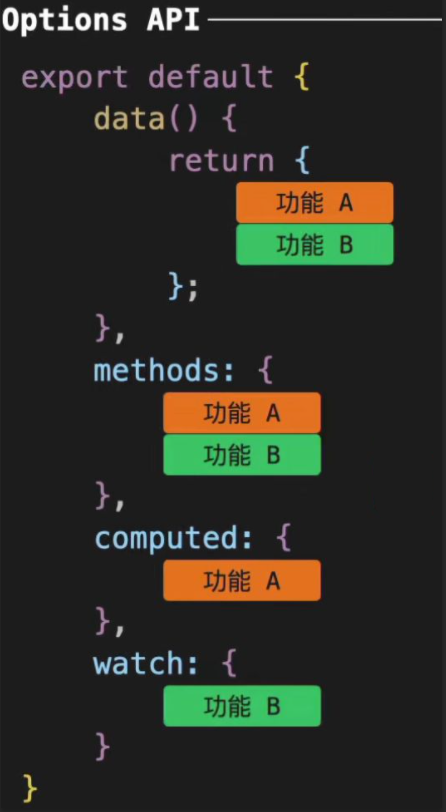
以上通过使用Composition Api都能迎刃而解

## 正文

### 一、Options Api

Options API，即大家常说的选项API，即以vue为后缀的文件，通过定义methods，computed，watch，data等属性与方法，共同处理页面逻辑

如下图：



可以看到Options代码编写方式，如果是组件状态，则写在data属性上，如果是方法，则写在methods属性上...

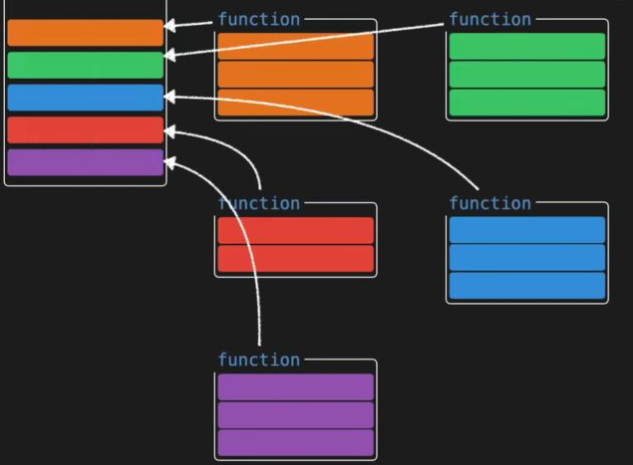
用组件的选项 (data、computed、methods、watch) 组织逻辑在大多数情况下都有效

然而，当组件变得复杂，导致对应属性的列表也会增长，这可能会导致组件难以阅读和理解

### 二、Composition Api

在 Vue3 Composition API 中，组件根据逻辑功能来组织的，一个功能所定义的所有 API 会放在一起（更加的高内聚，低耦合）

即使项目很大，功能很多，我们都能快速的定位到这个功能所用到的所有 API



### 三、对比

下面对Composition Api与Options Api进行两大方面的比较

* 逻辑组织
* 逻辑复用

#### 逻辑组织

##### Options API

假设一个组件是一个大型组件，其内部有很多处理逻辑关注点（对应下图不用颜色）



可以看到，这种碎片化使得理解和维护复杂组件变得困难

选项的分离掩盖了潜在的逻辑问题。此外，在处理单个逻辑关注点时，我们必须不断地“跳转”相关代码的选项块

##### Compostion API

而Compositon API正是解决上述问题，将某个逻辑关注点相关的代码全都放在一个函数里，这样当需要修改一个功能时，就不再需要在文件中跳来跳去

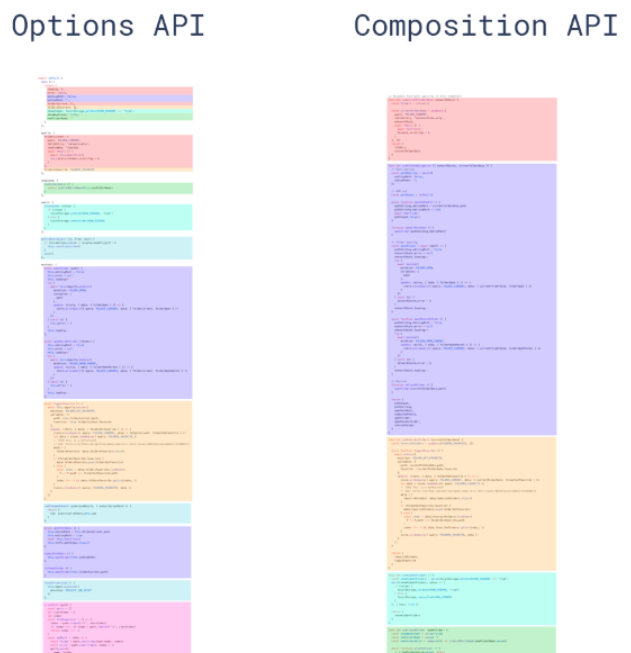
下面举个简单例子，将处理count属性相关的代码放在同一个函数了

function useCount() {  
 let count = ref(10);  
 let double = computed(() => {  
 return count.value \* 2;  
 });  
  
 const handleConut = () => {  
 count.value = count.value \* 2;  
 };  
  
 console.log(count);  
  
 return {  
 count,  
 double,  
 handleConut,  
 };  
}

组件上中使用count

export default defineComponent({  
 setup() {  
 const { count, double, handleConut } = useCount();  
 return {  
 count,  
 double,  
 handleConut  
 }  
 },  
});

再来一张图进行对比，可以很直观地感受到 Composition API在逻辑组织方面的优势，以后修改一个属性功能的时候，只需要跳到控制该属性的方法中即可



#### 逻辑复用

在Vue2中，我们是用过mixin去复用相同的逻辑

下面举个例子，我们会另起一个mixin.js文件

export const MoveMixin = {  
 data() {  
 return {  
 x: 0,  
 y: 0,  
 };  
 },  
  
 methods: {  
 handleKeyup(e) {  
 console.log(e.code);  
 // 上下左右 x y  
 switch (e.code) {  
 case "ArrowUp":  
 this.y--;  
 break;  
 case "ArrowDown":  
 this.y++;  
 break;  
 case "ArrowLeft":  
 this.x--;  
 break;  
 case "ArrowRight":  
 this.x++;  
 break;  
 }  
 },  
 },  
  
 mounted() {  
 window.addEventListener("keyup", this.handleKeyup);  
 },  
  
 unmounted() {  
 window.removeEventListener("keyup", this.handleKeyup);  
 },  
};

然后在组件中使用

<template>  
 <div>  
 Mouse position: x {{ x }} / y {{ y }}  
 </div>  
</template>  
<script>  
import mousePositionMixin from './mouse'  
export default {  
 mixins: [mousePositionMixin]  
}  
</script>

使用单个mixin似乎问题不大，但是当我们一个组件混入大量不同的 mixins 的时候

mixins: [mousePositionMixin, fooMixin, barMixin, otherMixin]

会存在两个非常明显的问题：

* 命名冲突
* 数据来源不清晰

现在通过Compositon API这种方式改写上面的代码

import { onMounted, onUnmounted, reactive } from "vue";  
export function useMove() {  
 const position = reactive({  
 x: 0,  
 y: 0,  
 });  
  
 const handleKeyup = (e) => {  
 console.log(e.code);  
 // 上下左右 x y  
 switch (e.code) {  
 case "ArrowUp":  
 // y.value--;  
 position.y--;  
 break;  
 case "ArrowDown":  
 // y.value++;  
 position.y++;  
 break;  
 case "ArrowLeft":  
 // x.value--;  
 position.x--;  
 break;  
 case "ArrowRight":  
 // x.value++;  
 position.x++;  
 break;  
 }  
 };  
  
 onMounted(() => {  
 window.addEventListener("keyup", handleKeyup);  
 });  
  
 onUnmounted(() => {  
 window.removeEventListener("keyup", handleKeyup);  
 });  
  
 return { position };  
}

在组件中使用

<template>  
 <div>  
 Mouse position: x {{ x }} / y {{ y }}  
 </div>  
</template>  
  
<script>  
import { useMove } from "./useMove";  
import { toRefs } from "vue";  
export default {  
 setup() {  
 const { position } = useMove();  
 const { x, y } = toRefs(position);  
 return {  
 x,  
 y,  
 };  
  
 },  
};  
</script>

可以看到，整个数据来源清晰了，即使去编写更多的 hook 函数，也不会出现命名冲突的问题

### 小结

* 在逻辑组织和逻辑复用方面，Composition API是优于Options API
* 因为Composition API几乎是函数，会有更好的类型推断。
* Composition API对 tree-shaking 友好，代码也更容易压缩
* Composition API中见不到this的使用，减少了this指向不明的情况
* 如果是小型组件，可以继续使用Options API，也是十分友好的