

3.Composition API与响应式系统



Composition API 是 Vue 3 最重要的新特性，也是面试中的高频考点。它不仅改变了我们组织代码的方式，还为逻辑复用提供了更优雅的解决方案。掌握它，你就掌握了 Vue 3 的精髓。

为什么需要 Composition API

概念

Composition API 的诞生是为了解决 **Options API** 在**大型、复杂组件**开发中遇到的痛点。

在 Options API 中，一个功能的实现逻辑（数据、方法、计算属性等）被迫分散在 `data`、`methods`、`computed` 等不同选项中。当组件变得复杂时，相关联的代码被拆分，而不相关的代码却聚合在一起。这导致开发者在理解和维护某个特定功能时，需要在单个文件中反复跳转，极大地降低了开发效率和代码的可读性。

Composition API 提出了一个核心思想：**根据逻辑功能来组织代码，而不是根据选项类型。**

代码示例

✗ Options API 的痛点：逻辑分散

在下面的示例中，`用户`、`文章` 和 `搜索` 这三个功能的逻辑分别散落在 `data`、`computed` 和 `methods` 中。

代码块

```
1  // 示例：一个包含用户、文章、搜索三个功能的复杂组件
2  export default {
3    // 1. 数据定义区
4    data() {
5      return {
6        // 用户相关数据
7        user: null,
8        userLoading: false,
9      }
```

```

10      // 文章相关数据
11      articles: [],
12      articlesLoading: false,
13
14      // 搜索相关数据
15      searchQuery: '',
16    }
17  },
18
19  // 2. 计算属性区
20  computed: {
21    // 用户相关计算属性
22    userFullName() {
23      /* ... */
24    },
25    // 文章相关计算属性
26    publishedArticles() {
27      /* ... */
28    },
29  },
30
31  // 3. 方法区
32  methods: {
33    // 用户相关方法
34    async fetchUser() { /* ... */ },
35    updateUser() { /* ... */ },
36
37    // 文章相关方法
38    async fetchArticles() { /* ... */ },
39    deleteArticle() { /* ... */ },
40
41    // 搜索相关方法
42    performSearch() { /* ... */ },
43  }
44  // 🤔 当需要修改“用户”功能时，你需要在 data, computed, methods 之间来回跳转。
45  }

```

✅ Composition API 的优势：逻辑聚合

使用 Composition API，我们可以将同一个功能的代码封装在独立的函数（称为 "Composable"）中，然后在 `setup` 函数中组合使用。

代码块

```
1 import { ref, computed } from 'vue'
```

```
2
3 // -----
4 // 功能一：用户管理 (User Logic)
5 // -----
6 function useUser() {
7   // 数据
8   const user = ref(null)
9   const userLoading = ref(false)
10
11   // 计算属性
12   const userFullName = computed(() => /* ... */)
13
14   // 方法
15   async function fetchUser() { /* ... */ }
16   function updateUser(data) { /* ... */ }
17
18   // 将所有与用户相关的 API 一起返回
19   return { user, userLoading, userFullName, fetchUser, updateUser }
20 }
21
22 // -----
23 // 功能二：文章管理 (Articles Logic)
24 // -----
25 function useArticles() {
26   const articles = ref([])
27   const articlesLoading = ref(false)
28   const publishedArticles = computed(() => /* ... */)
29   async function fetchArticles() { /* ... */ }
30   function deleteArticle(id) { /* ... */ }
31
32   return { articles, articlesLoading, publishedArticles, fetchArticles,
deleteArticle }
33 }
34
35
36 // -----
37 // 在组件中组合使用
38 // -----
39 export default {
40   setup() {
41     // 引入用户功能
42     const { user, userFullName, fetchUser } = useUser()
43     // 引入文章功能
44     const { articles, publishedArticles, fetchArticles } = useArticles()
45
46     // 😊 相关逻辑聚合，不相关逻辑分离，代码组织清晰！
47
```

```
48      // 将需要暴露给模板的变量和方法返回
49      return {
50          user,
51          userFullName,
52          articles,
53          publishedArticles,
54          fetchUser,
55          fetchArticles
56      }
57  }
58 }
```

深入解析

Composition API 带来了思维模式的转变：

1. **更优的逻辑复用**：通过**组合函数 (Composables)** 来复用逻辑，取代了 `mixins`。`Composables` 是普通的 JavaScript 函数，解决了 `mixins` 的命名冲突、数据来源不清晰和隐式依赖等问题。
2. **更好的类型推断**：代码基于普通的函数和变量，TypeScript 可以完美地进行类型推断，解决了 Options API 中 `this` 的类型推断难题。
3. **更灵活的代码组织**：你可以将复杂逻辑拆分到多个独立的文件中，让大型项目维护起来更加轻松。
4. **更小的打包体积**：Composition API 的函数是按需导入的，例如 `ref`，`computed` 等。没有用到的代码可以被构建工具进行 "Tree-shaking"，从而减小最终的打包体积。

面试建议

问："为什么 Vue 3 要引入 Composition API？它解决了什么问题？"

回答要点：

1. **先肯定 Options API**："Options API 在中小型项目中非常直观易懂，这是 Vue 易学性的重要体现。"
2. **再指出其痛点**："然而，在开发大型复杂组件时，Options API 会导致**逻辑关注点分散**，相关代码被拆分到 `data`，`methods` 等不同选项中，难以维护。同时，依赖 `mixins` 的逻辑复用方案存在命名冲突和数据来源不清晰等问题，且对 TypeScript 的类型推断不够友好。"
3. **最后引出解决方案**："Composition API 正是为了解决这些问题而生的。它允许我们通过**组合函数**的方式，将**相关逻辑聚合**在一起，实现了更清晰、更可维护、类型更安全的代码组织和逻辑复用模式。"

4. **补充说明：**需要强调的是，Composition API 是对 Options API 的**补充而非替代**，开发者可以根据项目复杂度和团队习惯灵活选择。

核心 API 详解

1. `ref()` - 创建基础类型的响应式引用

`ref` 用于将一个基础类型的值（如 `string`，`number`，`boolean`）包装成一个响应式的对象。

基本用法

代码块

```
1  // MyComponent.vue
2
3  import { ref } from 'vue'
4
5  export default {
6    setup() {
7      // ref() 返回一个包含 .value 属性的响应式对象
8      const count = ref(0)
9      const message = ref('Hello Vue 3')
10
11     // 在 <script> 中，必须通过 .value 访问或修改其值
12     console.log(count.value) // 输出: 0
13
14     const increment = () => {
15       count.value++
16     }
17
18     // 将 ref 对象返回，使其在模板中可用
19     return {
20       count,
21       message,
22       increment
23     }
24   }
25 }
```

```

1 <template>
2   <!-- 在模板中使用时, Vue 会自动“解包” (unwrap), 无需 .value -->
3   <div>{{ count }}</div>           <!-- 渲染: 0 -->
4   <div>{{ message }}</div>       <!-- 渲染: Hello Vue 3 -->
5   <button @click="increment">+1</button>
6 </template>

```



面试标准回答

`ref` 用于创建一个响应式的引用，通常用于**基本类型**数据。它返回一个对象，这个对象只有一个 `.value` 属性，指向内部的值。在 JavaScript 逻辑中，我们必须通过 `.value` 来访问和修改这个值；但在 Vue 模板中，Vue 会自动解包，我们可以直接使用变量名。`ref` 的响应式能力是基于 `Proxy` 实现的，当 `.value` 被修改时，会触发所有依赖它的更新。

2. `reactive()` - 创建对象的响应式代理

`reactive` 用于创建一个对象的**深度响应式**代理。

基本用法

代码块

```

1  import { reactive } from 'vue'
2
3  export default {
4    setup() {
5      // reactive() 接收一个对象, 并返回其响应式代理
6      const state = reactive({
7        count: 0,
8        user: {
9          name: 'John',
10         age: 25
11       },
12       items: []
13     })
14
15     const updateUser = () => {
16       // 可以直接修改属性, 无需 .value
17       state.user.name = 'Jane'
18       state.items.push({ id: 1, title: 'New Item' })

```

```

19     }
20
21     // 返回整个 state 对象
22     return {
23         state,
24         updateUser
25     }
26 }
27 }

```

在模板中使用时，可以通过 `state.user.name` 的方式访问。

ref vs reactive 选择指南

- **ref**：推荐用于处理**基本数据类型**（String, Number, Boolean）。
- **reactive**：推荐用于处理**对象或数组**。

代码块

```

1  // ✅ 推荐用法，代码语义更清晰
2  const count = ref(0);
3  const user = reactive({ name: 'John' });
4
5  // ❌ 不推荐
6  const countAsObject = reactive({ value: 0 }); // 用 reactive 包装基本类型，过于繁琐
7  const userAsRef = ref({ name: 'John' });      // 用 ref 包装对象，每次访问都需要 .value (userAsRef.value.name)

```



面试标准回答

reactive 用于创建对象的响应式代理，它会**深度地遍历**对象所有属性，将其转换为 **Proxy**。与 **ref** 不同，**reactive** 返回的代理对象可以直接访问和修改其属性，无需 **.value**。

选择原则：坚持“基本类型用 **ref**，对象类型用 **reactive**”的约定，可以使代码的意图更加清晰，提高可读性。

3. `computed()` - 计算属性

`computed` 用于创建一个根据其他响应式数据派生出来的值。它具有缓存特性。

基本用法

代码块

```
1  import { ref, computed } from 'vue'
2
3  export default {
4    setup() {
5      const firstName = ref('John')
6      const lastName = ref('Doe')
7
8      // 1. 只读的计算属性 (传入一个 getter 函数)
9      // 只有当 firstName.value 或 lastName.value 变化时才会重新计算
10     const fullName = computed(() => `${firstName.value} ${lastName.value}`)
11
12     // 2. 可写的计算属性 (传入一个包含 get 和 set 的对象)
13     const editableName = computed({
14       get: () => `${firstName.value} ${lastName.value}`,
15       set: (newValue) => {
16         // 当你尝试给 editableName.value 赋值时, set 函数会被调用
17         const names = newValue.split(' ')
18         firstName.value = names[0]
19         lastName.value = names[1] || ''
20       }
21     })
22
23     return {
24       fullName,
25       editableName
26     }
27   }
28 }
```



面试标准回答

`computed` 用于创建计算属性，它会**基于其响应式依赖进行缓存**。这意味着只要依赖项没有发生变化，多次访问计算属性会立即返回之前计算过的结果，而不会重新执行计算函数。这是它与 `methods` 的核心区别。`computed` 可以是只读的，也可以通过提供 `get` 和 `set` 函数来创建可写的计算属性。

4. watch() & watchEffect() - 侦听器

watch：明确指定侦听源

watch 需要明确指定要侦听的数据源，并在数据变化时执行回调函数。

代码块

```
1  import { ref, reactive, watch } from 'vue'
2
3  export default {
4    setup() {
5      const count = ref(0)
6      const state = reactive({ name: 'John' })
7
8      // 1. 侦听单个 ref
9      watch(count, (newVal, oldVal) => {
10        console.log(`count 从 ${oldVal} 变为 ${newVal}`)
11      })
12
13      // 2. 侦听 getter 函数，可以访问对象属性
14      watch(() => state.name, (newName, oldName) => {
15        console.log(`name 变化了: ${newName}`)
16      })
17
18      // 3. 侦听多个源
19      watch([count, () => state.name], ([newCount, newName], [oldCount,
oldName]) => {
20        console.log('count 或 name 发生了变化')
21      })
22
23      // 4. 选项: deep (深度侦听) 和 immediate (立即执行)
24      watch(
25        () => state,
26        (newState, oldState) => { console.log('state 深度变化') },
27        { deep: true, immediate: true } // immediate 会让 watch 在初始化时立即执行一
次
28      )
29
30      return { count, state }
31    }
32  }
```

watchEffect：自动追踪依赖

`watchEffect` 会立即执行一次，并**自动追踪**回调函数中所有使用到的响应式依赖。当任何依赖发生变化时，它会重新运行。

代码块

```
1  import { ref, watchEffect } from 'vue'
2
3  export default {
4    setup() {
5      const count = ref(0)
6      const message = ref('hello')
7
8      // watchEffect 会立即执行，并自动追踪 count 和 message 的变化
9      watchEffect(() => {
10        // 在这个函数体内用到了 count.value 和 message.value
11        // 所以 Vue 会自动侦听这两个响应式引用的变化
12        console.log(`count: ${count.value}, message: ${message.value}`)
13      })
14
15      // 它等价于一个立即执行 (immediate: true) 且自动依赖收集的 watch
16
17      return { count, message }
18    }
19  }
```

对比

特性	<code>watch</code>	<code>watchEffect</code>
侦听源	需要手动指定	自动追踪依赖，无需指定
执行时机	默认在数据变化后执行	立即执行一次，然后依赖变化后执行
回调参数	可以访问新值和旧值	无法访问旧值
使用场景	需要精确控制侦听目标，或需要访问旧值时	逻辑简单，只需依赖变化就执行副作用时



面试标准回答

`watch` 和 `watchEffect` 都用于在数据变化时执行副作用。主要区别在于：`watch` 需要显式指定侦听的数据源，让你能精确控制副作用的触发时机，并且可以访问到新值和旧

值。而 `watchEffect` 会自动收集其回调函数中的响应式依赖，并在初始化时立即执行一次，代码更简洁，但无法访问旧值。

响应式陷阱与解决方案

陷阱 1：解构 `reactive` 对象导致失去响应性

直接解构 `reactive` 对象会使其属性变为普通变量，从而失去响应性。

代码块

```
1  import { reactive, toRefs } from 'vue'
2
3  export default {
4    setup() {
5      const state = reactive({ count: 0, name: 'John' })
6
7      // ❌ 错误: count 和 name 只是普通变量，不再具有响应性
8      // 当 state.count 改变时，这里的 count 不会更新
9      const { count, name } = state
10
11     // ✅ 正确: 使用 toRefs()
12     // toRefs 会将 reactive 对象的每个属性都转换为一个 ref
13     const stateAsRefs = toRefs(state)
14     // 现在 stateAsRefs.count 和 stateAsRefs.name 都是 ref，可以在模板中安全使用
15
16     return {
17       // 如果直接返回 count，它不是响应式的
18       // 必须返回 toRefs 转换后的结果
19       ...stateAsRefs // 使用扩展运算符将 { count: ref, name: ref } 返回
20     }
21   }
22 }
```

陷阱 2：错误地替换响应式对象

对于 `ref` 包裹的对象或数组，修改时必须通过 `.value` 属性进行。

代码块

```
1  import { ref } from 'vue'
2
3  export default {
4    setup() {
5      let list = ref([1, 2, 3])
6
7      const updateList = () => {
8        // ❌ 错误: 这会使 list 变量指向一个新的普通数组, 原来的响应式引用丢失了
9        // list = [4, 5, 6]
10
11        // ✅ 正确: 始终通过 .value 来修改 ref 的值
12        list.value = [4, 5, 6]
13      }
14      return { list, updateList }
15    }
16  }
```

与 React Hooks 对比

特性	Vue Composition API	React Hooks
执行时机	<code>setup</code> 函数只在组件创建时执行一次。	Hooks 函数在每次组件渲染时都会执行。
响应式系统	基于 Proxy 的响应式系统，数据变更 自动触发 更新。	需要通过 <code>useState</code> 的 <code>set</code> 函数手动更新状态，触发重渲染。
依赖管理	<code>computed</code> 和 <code>watchEffect</code> 自动追踪 依赖，无需手动声明。	<code>useMemo</code> , <code>useCallback</code> 需要手动管理 依赖数组，否则可能导致性能问题或陈旧闭包。
心智负担	心智负担较低，更接近原生 JavaScript 的编程体验。	心智负担较高，需要理解闭包、依赖数组等规则。

高级应用：自定义组合函数

组合函数是 Composition API 的精髓，它允许我们将可复用的有状态逻辑封装起来。

示例：设计一个 `useFetch` 组合函数

代码块

```
1  // composables/useFetch.js
2
3  import { ref, watchEffect, toValue } from 'vue'
4
5  // 一个组合函数就是一个返回响应式状态的普通函数
6  export function useFetch(url) {
7    const data = ref(null)
8    const error = ref(null)
9    const loading = ref(true)
10
11    const fetchData = async () => {
12      // 重置状态
13      loading.value = true
14      error.value = null
15
16      try {
17        // toValue 可以将 ref 或 getter 解包为普通值
18        const response = await fetch(toValue(url))
19        if (!response.ok) throw new Error('Network response was not ok')
20        data.value = await response.json()
21      } catch (e) {
22        error.value = e
23      } finally {
24        loading.value = false
25      }
26    }
27
28    // 使用 watchEffect 侦听 URL 的变化
29    // 如果 url 是一个 ref (例如 computed), 当它变化时会自动重新 fetch
30    watchEffect(fetchData)
31
32    // 返回响应式状态和方法
33    return { data, error, loading, refetch: fetchData }
34  }
```

使用示例

代码块

```
1  // MyComponent.vue
```

```
2
3 import { ref, computed } from 'vue'
4 import { useFetch } from '../composables/useFetch'
5
6 export default {
7   setup() {
8     const userId = ref(1)
9
10    // 创建一个计算属性作为 URL，当 userId 变化时，URL 会自动更新
11    const url = computed(() => `https://api.example.com/users/${userId.value}`)
12
13    // 使用自定义的 useFetch Hook
14    // 当 url 变化时，useFetch 内部的 watchEffect 会自动重新请求数据
15    const { data: user, loading, error } = useFetch(url)
16
17    const changeUser = () => {
18      userId.value++ // 这会触发 url 的重新计算，进而触发 useFetch 重新执行
19    }
20
21    return { user, loading, error, changeUser }
22  }
23 }
```