4.Vue 3 生命周期与副作用管理

✏️ Vue 3 的生命周期和副作用管理是面试的高频考点。深刻理解它们可以帮助你写出健壮、可维护的代码，展现你的技术深度。

1. Vue 3 生命周期：新变化与核心理念

1.1 核心变化

Vue 3 对生命周期系统进行了重要调整，核心变化体现在 **Composition API** 中：

1. **全面函数化**：所有生命周期钩子都变成了可以在 `setup()` 中调用的函数 (例如 `onMounted`，`onUpdated`)。这使得相关逻辑可以轻松地抽离到独立的组合函数中，极大提高了复用性。
2. **命名统一**：废弃了 `beforeDestroy` 和 `destroyed`，统一为 `onBeforeUnmount` 和 `onUnmounted`，更准确地描述了组件“卸载”的过程。
3. **`setup` 的中心地位**：`setup` 函数在组件实例创建之前执行，它在时机上包揽了 Vue 2 中的 `beforeCreate` 和 `created`，成为组件初始化的核心入口。
4. **新增调试钩子**：增加了 `onRenderTracked` 和 `onRenderTriggered`，用于在开发模式下追踪和调试组件的响应式依赖。

1.2 代码对比：Options API vs Composition API

Options API (Vue 2/3)

代码块

```
// 逻辑分散在不同的选项中
export default {
  data() {
    return {
      message: 'Hello',
      timer: null
    }
  },
  mounted() {
    console.log('DOM 挂载完成');
    this.timer = setInterval(() => {
      console.log('定时任务');
    }, 1000);
  },
```

```
  beforeUnmount() { // Vue 2 中是 beforeDestroy
    console.log('组件即将卸载');
    if (this.timer) {
      clearInterval(this.timer);
    }
  }
}
```

✅ Composition API (Vue 3)

代码块

```
import { ref, onMounted, onBeforeUnmount } from 'vue';

// 所有相关逻辑都聚合在 setup 中
export default {
  setup() {
    const message = ref('Hello');
    let timer = null;

    // 生命周期钩子作为函数调用
    onMounted(() => {
      console.log('DOM 挂载完成');
      timer = setInterval(() => {
        console.log('定时任务');
      }, 1000);
    });

    // 清理逻辑也在这里，代码关联性更强
    onBeforeUnmount(() => {
      console.log('组件即将卸载');
      if (timer) {
        clearInterval(timer);
      }
    });

    return { message };
  }
}
```

1.3 生命周期钩子映射表

Vue 2 Options API	Vue 3 Composition API	执行时机	典型用途
`beforeCreate`	`setup()`	实例初始化前	基本被 `setup` 替代
`created`	`setup()`	实例创建后	初始化数据、发起网络请求
`beforeMount`	`onBeforeMount`	DOM 挂载前	准备工作，访问不到 DOM
`mounted`	`onMounted`	DOM 挂载后	DOM 操作、集成第三方库
`beforeUpdate`	`onBeforeUpdate`	数据更新，DOM 重新渲染前	获取更新前的 DOM 状态
`updated`	`onUpdated`	DOM 更新后	执行依赖于 DOM 更新的操作
`beforeDestroy`	`onBeforeUnmount`	组件实例卸载前	清理定时器、事件监听等副作用
`destroyed`	`onUnmounted`	组件实例卸载后	最后的清理工作

---

2. 副作用（Side Effects）管理

副作用是指**组件中会影响外部环境的操作**，如网络请求、DOM 操作、定时器和事件监听等。妥善管理副作用是保证应用稳定性的关键。

2.1 使用生命周期钩子管理

最基础的副作用管理方式就是利用生命周期钩子，在合适的时机创建和销毁副作用。

- **`onMounted`**：组件挂载到 DOM 后执行，适合执行需要访问 DOM 的操作或一次性的初始化任务。
- **`onBeforeUnmount`** / **`onUnmounted`**：组件卸载前/后执行，是清理副作用（如定时器、全局事件监听）的理想位置，防止内存泄漏。

代码块

```
import { ref, onMounted, onBeforeUnmount } from 'vue';

export default {
  setup() {
```

```
    let timer = null;
    const handleResize = () => { /* ... */ };

    // 在 onMounted 中设置副作用
    onMounted(() => {
      const element = document.querySelector('#my-element');
      if (element) element.focus(); // DOM 操作

      timer = setInterval(() => console.log('tick'), 1000); // 定时器
      window.addEventListener('resize', handleResize); // 事件监听
    });

    // 在 onBeforeUnmount 中清理副作用
    onBeforeUnmount(() => {
      clearInterval(timer);
      window.removeEventListener('resize', handleResize);
    });

    return {};
  }
}
```

- **`onUpdated`**：在组件因响应式数据变化而更新 DOM 后调用。适用于需要操作更新后 DOM 的场景，例如聊天窗口滚动到底部。

代码块

```
import { ref, onUpdated, nextTick } from 'vue';

export default {
  setup() {
    const list = ref([]);
    const scrollContainer = ref(null); // <div ref="scrollContainer"></div>

    onUpdated(() => {
      // DOM 更新后，将滚动条滚动到底部
      if (scrollContainer.value) {
        scrollContainer.value.scrollTop = scrollContainer.value.scrollHeight;
      }
    });

    // 注意：onUpdated 在每次更新后都会执行。
    // 如果想基于特定数据的变化来执行操作，`watch` 是更好的选择。

    return { list, scrollContainer };
```

```
  }
}
```

2.2 使用 Watchers 精确管理

对于需要响应特定数据变化的副作用，Vue 提供了 `watch` 和 `watchEffect`，它们提供了比生命周期钩子更精确的控制力。

`watchEffect`：自动追踪依赖

`watchEffect` 会立即执行一次，然后自动追踪其回调函数中所有使用到的响应式依赖。当任何一个依赖变化时，它会重新运行。

- **优点**：简单直接，无需手动指定依赖。
- **适用场景**：当副作用的依赖关系复杂或不确定时。

代码块

```
import { ref, watchEffect } from 'vue';

export default {
  setup() {
    const userId = ref('1');
    const userData = ref(null);

    // watchEffect 会自动追踪 userId.value 的变化
    watchEffect(async (onInvalidate) => {
      if (!userId.value) return;

      const controller = new AbortController();
      // onInvalidate 注册一个清理函数，在副作用重新执行或组件卸载前调用
      onInvalidate(() => controller.abort());

      try {
        userData.value = await fetch(`/api/users/${userId.value}`, { signal: controller.signal });
      } catch (error) {
        if (error.name !== 'AbortError') {
          console.error('Failed to fetch user data:', error);
        }
      }
    });

    return { userId, userData };
  }
```

```
27  }
```

`watch`：明确指定依赖

`watch` 让你明确指定要监听的一个或多个数据源，并在它们变化时执行回调。

- **优点**：控制更精确，可以访问新值和旧值，并且可以通过选项（`deep`，`immediate`）进行深度监听或立即执行。
- **适用场景**：当你想在特定数据变化时执行逻辑，或者需要旧数据进行比较时。

代码块

```
1   import { ref, watch } from 'vue';
2
3   export default {
4     setup() {
5       const keyword = ref('');
6       const results = ref([]);
7
8       // 1. 监听单个 ref
9       watch(keyword, async (newVal, oldVal) => {
10        if (newVal.length > 1) {
11          results.value = await fetch(`/api/search?q=${newVal}`);
12        } else {
13          results.value = [];
14        }
15      });
16
17      const filters = ref({ price: 100, category: 'books' });
18
19      // 2. 深度监听对象
20      watch(
21        filters,
22        (newFilters) => {
23          // 当 filters 内部属性变化时执行
24          console.log('Filters changed:', newFilters);
25        },
26        { deep: true } // 必须开启 deep 选项
27      );
28
29      return { keyword, results, filters };
30    }
31  }
```

3. 与 React `useEffect` 对比

Vue 的 `watch` / `watchEffect` 与 React 的 `useEffect` 目的相似，但心智模型不同。

特性	Vue (`watch` / `watchEffect`)	React (`useEffect`)
依赖追踪	`watchEffect` 自动追踪，`watch` 手动指定	总是需要手动在依赖数组中指定
执行时机	默认在数据变化后、DOM 更新前执行	在组件完成渲染后执行
心智负担	较低，自动追踪不易出错	较高，容易忘记添加依赖，导致 bug
性能	响应式系统实现细粒度更新，副作用精准触发	组件级重渲染，依赖 `useCallback`，`useMemo` 优化

代码对比：实现相同功能

Vue (`watchEffect`)

代码块

```
// Vue: 自动追踪 userId 的变化
import { ref, watchEffect } from 'vue';

const userId = ref(1);
const user = ref(null);

watchEffect(async (onInvalidate) => {
  const controller = new AbortController();
  onInvalidate(() => controller.abort());
  user.value = await fetchUser(userId.value, controller.signal);
});
```

React (`useEffect`)

```
代码块
1  // React: 必须在依赖数组中手动指定 userId
2  import { useState, useEffect } from 'react';
3
4  const [userId, setUserId] = useState(1);
5  const [user, setUser] = useState(null);
6
7  useEffect(() => {
8    const controller = new AbortController();
9    fetchUser(userId, controller.signal).then(setUser);
10
11   return () => controller.abort(); // 返回清理函数
12 }, [userId]); // 手动指定依赖数组
```

---

4. 最佳实践与常见陷阱

✅ 最佳实践

1. **逻辑聚合**：将创建副作用（如 `setInterval`）和清理它的逻辑（`clearInterval`）放在一起，最好使用 `onMounted` 和 `onBeforeUnmount` 配对。
2. **条件性执行**：在 `watch` 或 `watchEffect` 内部添加条件判断，避免在不必要时（如搜索词太短）执行昂贵的操作。
3. **优先选择 `watch`**：当你明确知道副作用依赖哪个状态时，优先使用 `watch`，因为它的意图更清晰。当依赖关系复杂或不确定时，再考虑 `watchEffect`。

❌ 常见陷阱

1. **忘记清理**：最常见的错误是在组件卸载时忘记清理定时器、事件监听器或 WebSocket 连接，这会导致严重的内存泄漏。

```
代码块
1  // ❌ 错误：忘记在 onBeforeUnmount 中清理
2  onMounted(() => {
3    window.addEventListener('scroll', handleScroll);
```

```
});

// ✅ 正确
onMounted(() => window.addEventListener('scroll', handleScroll));
onBeforeUnmount(() => window.removeEventListener('scroll', handleScroll));
```

2. **在错误的生命周期访问 DOM**：在 `setup` 中直接访问 DOM 会失败，因为此时组件尚未挂载。所有 DOM 操作都应在 `onMounted` 之后进行。

代码块

```
// ❌ 错误：setup 执行时 DOM 不存在
setup() {
  const el = document.getElementById('my-element'); // el is null
}

// ✅ 正确
setup() {
  onMounted(() => {
    const el = document.getElementById('my-element'); // el is available
  });
}
```

---

5. 面试核心问题

Q1: "Vue 3 的生命周期相比 Vue 2 有什么核心变化？"

回答要点：

1. **从选项到函数**：最大的变化是在 Composition API 中，生命周期从 Options API 的对象属性变成了需要导入的函数，如 `onMounted`。
2. **`setup` 的整合**：`setup` 函数在时机上替代了 `beforeCreate` 和 `created`，成为组件初始化的入口。
3. **命名变更**：`beforeDestroy` 和 `destroyed` 被更名为了 `onBeforeUnmount` 和 `onUnmounted`，语义更清晰。
4. **优势**：这种函数式的转变让逻辑组织更灵活，可以轻松地将相关联的副作用逻辑（如创建和清理）聚合在一起，并通过组合式函数（Composables）实现复用。

Q2: "`watch` 和 `watchEffect` 有什么区别？应该如何选择？"

回答要点：

1. **依赖追踪：**
   - `watchEffect`：**自动追踪**。它会自动收集其回调函数中访问到的所有响应式数据作为依赖。
   - `watch`：**手动指定**。你必须明确地告诉它要监听哪个数据源。
2. **执行时机：**
   - `watchEffect`：**立即执行**一次，然后等待依赖变化后再次执行。
   - `watch`：默认是**懒执行**的，只有当被监听的数据源变化时才执行。可以通过 `{ immediate: true }` 选项使其立即执行。
3. **访问旧值：**
   - `watchEffect`：无法访问变化前的值。
   - `watch`：可以同时访问新值和旧值，方便进行比较。

如何选择：

- **用** `watch`：当你想精确控制监听目标，或者当副作用逻辑需要依赖旧值时。这是更常见的选择。
- **用** `watchEffect`：当副作用的依赖项很多，或者依赖关系不那么直观时，让 Vue 自动追踪会更方便。