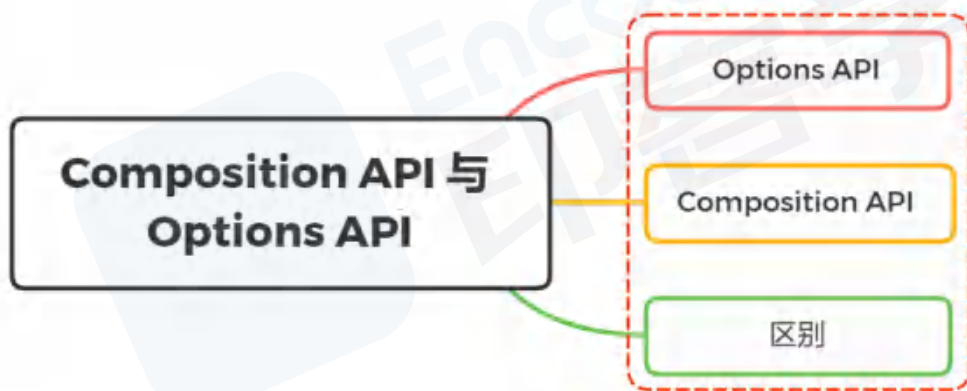


Vue3面试真题（6题）

1. Vue3.0 所采用的 Composition Api 与 Vue2.x 使用的 Options Api 有什么不同？



1.1. 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

1.2. 正文

1.2.1. 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：

Options API

```
export default {  
  data() {  
    return {  
      功能 A  
      功能 B  
    }  
  },  
  methods: {  
    功能 A  
    功能 B  
  },  
  computed: {  
    功能 A  
  },  
  watch: {  
    功能 B  
  }  
}
```

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...

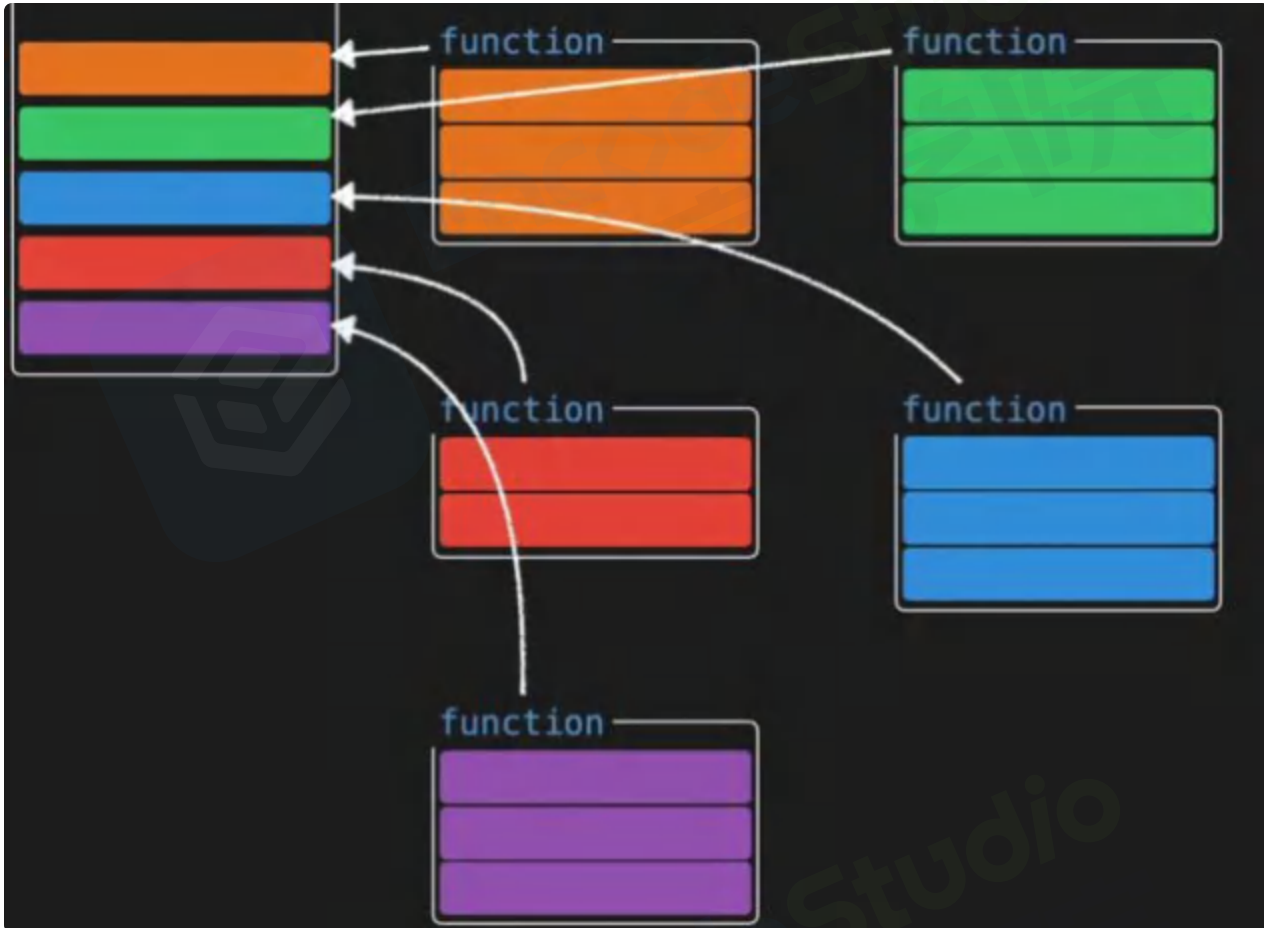
用组件的选项（`data`、`computed`、`methods`、`watch`）组织逻辑在大多数情况下都有效

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

1.2.2. 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



1.2.3. 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

1.2.3.1. 逻辑组织

1.2.3.1.1. 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

1.2.3.1.2. Compositon 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

```
JavaScript | 复制代码

1 function useCount() {
2   let count = ref(10);
3   let double = computed(() => {
4     return count.value * 2;
5   });
6
7   const handleConut = () => {
8     count.value = count.value * 2;
9   };
10
11   console.log(count);
12
13   return {
14     count,
15     double,
16     handleConut,
17   };
18 }
```

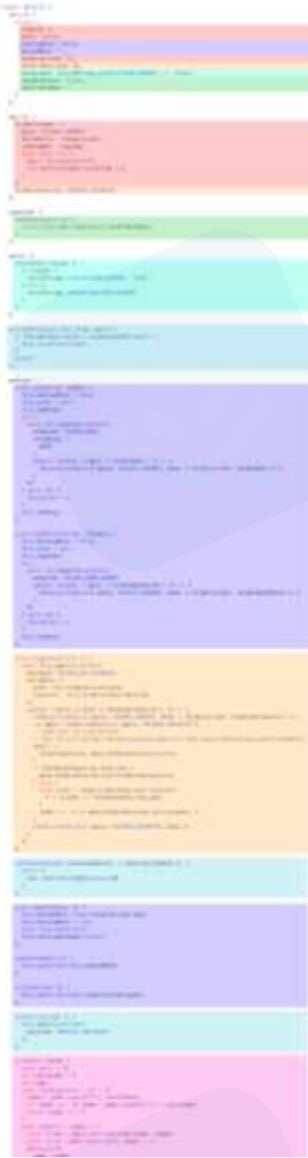
组件上中使用 `count`

```
JavaScript | 复制代码

1 export default defineComponent({
2   setup() {
3     const { count, double, handleConut } = useCount();
4     return {
5       count,
6       double,
7       handleConut
8     };
9   },
10 });
```

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

Options API



Composition API



1.2.3.2. 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

```
1 export const MoveMixin = {
2   data() {
3     return {
4       x: 0,
5       y: 0,
6     };
7   },
8
9   methods: {
10    handleKeyup(e) {
11      console.log(e.code);
12      // 上下左右 x y
13      switch (e.code) {
14        case "ArrowUp":
15          this.y--;
16          break;
17        case "ArrowDown":
18          this.y++;
19          break;
20        case "ArrowLeft":
21          this.x--;
22          break;
23        case "ArrowRight":
24          this.x++;
25          break;
26      }
27    },
28  },
29
30  mounted() {
31    window.addEventListener("keyup", this.handleKeyup);
32  },
33
34  unmounted() {
35    window.removeEventListener("keyup", this.handleKeyup);
36  },
37 };
```

然后在组件中使用

JavaScript | 复制代码

```
1 <template>
2   <div>
3     Mouse position: x {{ x }} / y {{ y }}
4   </div>
5 </template>
6 <script>
7   import mousePositionMixin from './mouse'
8   export default {
9     mixins: [mousePositionMixin]
10  }
11 </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

JavaScript | 复制代码

```
1 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


```
1  import { onMounted, onUnmounted, reactive } from "vue";
2  export function useMove() {
3    const position = reactive({
4      x: 0,
5      y: 0,
6    });
7
8    const handleKeyup = (e) => {
9      console.log(e.code);
10     // 上下左右 x y
11     switch (e.code) {
12       case "ArrowUp":
13         // y.value--;
14         position.y--;
15         break;
16       case "ArrowDown":
17         // y.value++;
18         position.y++;
19         break;
20       case "ArrowLeft":
21         // x.value--;
22         position.x--;
23         break;
24       case "ArrowRight":
25         // x.value++;
26         position.x++;
27         break;
28     }
29   };
30
31   onMounted(() => {
32     window.addEventListener("keyup", handleKeyup);
33   });
34
35   onUnmounted(() => {
36     window.removeEventListener("keyup", handleKeyup);
37   });
38
39   return { position };
40 }
```

在组件中使用

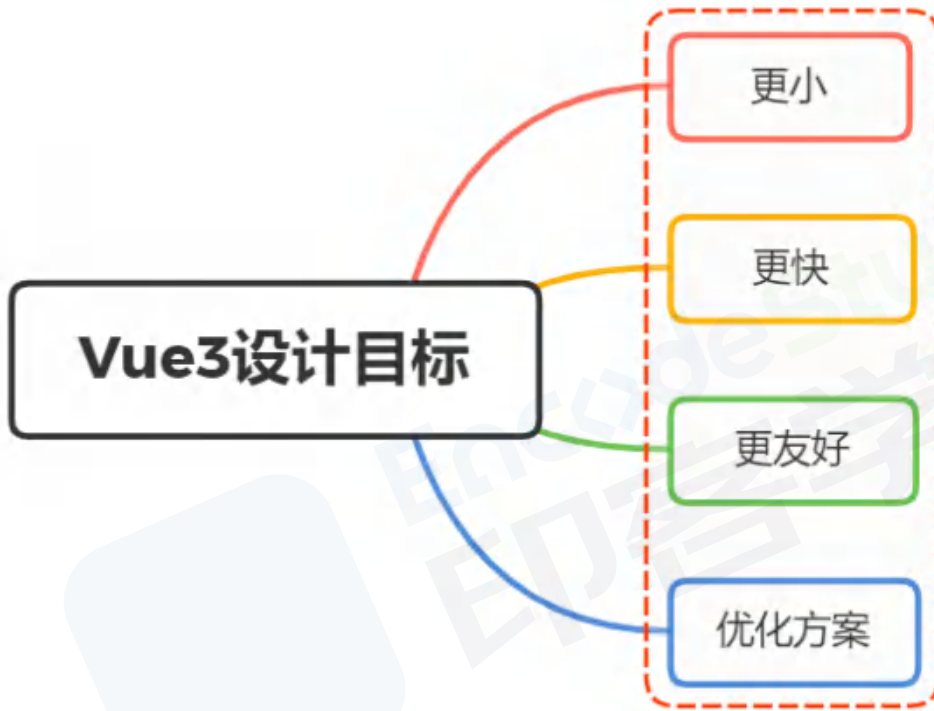
```
1 <template>
2   <div>
3     Mouse position: x {{ x }} / y {{ y }}
4   </div>
5 </template>
6
7 <script>
8   import { useMove } from "../useMove";
9   import { toRefs } from "vue";
10  export default {
11    setup() {
12      const { position } = useMove();
13      const { x, y } = toRefs(position);
14      return {
15        x,
16        y,
17      };
18    },
19  };
20 </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

1.3. 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

2. Vue3.0的设计目标是什么？做了哪些优化



2.1. 设计目标

不以解决实际业务痛点的更新都是耍流氓，下面我们来列举一下 `Vue3` 之前我们或许会面临的问题

- 随着功能的增长，复杂组件的代码变得越来越难以维护
- 缺少一种比较「干净」的在多个组件之间提取和复用逻辑的机制
- 类型推断不够友好
- `bundle` 的时间太久了

而 `Vue3` 经过长达两三年时间的筹备，做了哪些事情？

我们从结果反推

- 更小
- 更快
- TypeScript支持
- API设计一致性
- 提高自身可维护性
- 开放更多底层功能

一句话概述，就是更小更快更友好了

2.1.1. 更小

Vue3 移除一些不常用的 API

引入 tree-shaking，可以将无用模块“剪辑”，仅打包需要的，使打包的整体体积变小了

2.1.2. 更快

主要体现在编译方面：

- diff算法优化
- 静态提升
- 事件监听缓存
- SSR优化

2.1.3. 更友好

vue3 在兼顾 vue2 的 options API 的同时还推出了 composition API，大大增加了代码的逻辑组织和代码复用能力

这里代码简单演示下：

存在一个获取鼠标位置的函数

JavaScript | 复制代码

```
1  import { toRefs, reactive } from 'vue';
2  function useMouse(){
3      const state = reactive({x:0,y:0});
4      const update = e=>{
5          state.x = e.pageX;
6          state.y = e.pageY;
7      }
8      onMounted(()=>{
9          window.addEventListener('mousemove',update);
10     })
11     onUnmounted(()=>{
12         window.removeEventListener('mousemove',update);
13     })
14
15     return toRefs(state);
16 }
```

我们只需要调用这个函数，即可获取 x、y 的坐标，完全不用关注实现过程

试想一下，如果很多类似的第三方库，我们只需要调用即可，不必关注实现过程，开发效率大大提高

同时，`VUE3` 是基于 `typescript` 编写的，可以享受到自动的类型定义提示

2.2. 优化方案

`vue3` 从很多层面都做了优化，可以分成三个方面：

- 源码
- 性能
- 语法 API

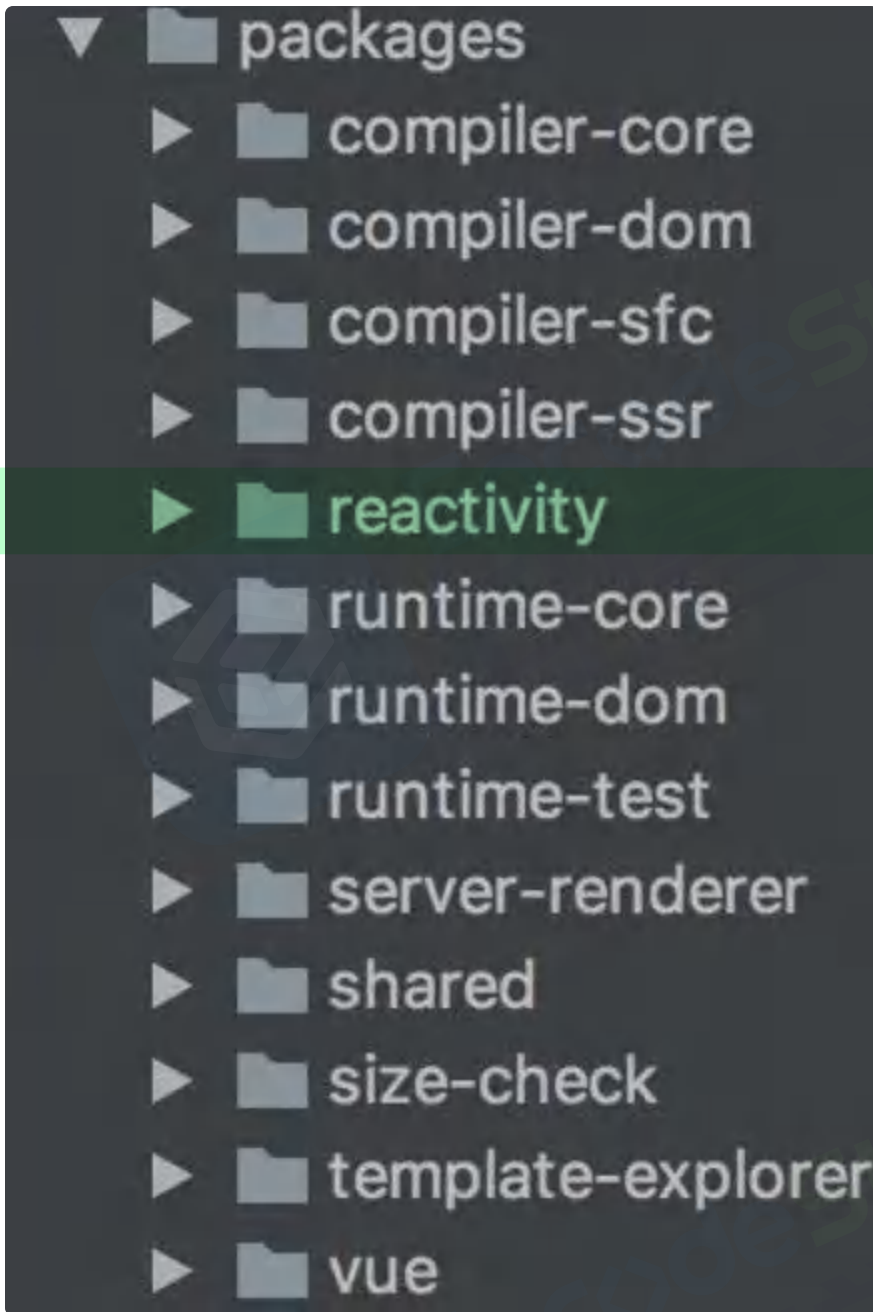
2.2.1. 源码

源码可以从两个层面展开：

- 源码管理
- TypeScript

2.2.1.1. 源码管理

`vue3` 整个源码是通过 `monorepo` 的方式维护的，根据功能将不同的模块拆分到 `packages` 目录下面不同的子目录中



这样使得模块拆分更细化，职责划分更明确，模块之间的依赖关系也更加明确，开发人员也更容易阅读、理解和更改所有模块源码，提高代码的可维护性

另外一些 `package`（比如 `reactivity` 响应式库）是可以独立于 `Vue` 使用的，这样用户如果只想使用 `Vue3` 的响应式能力，可以单独依赖这个响应式库而不用去依赖整个 `Vue`

2.2.1.2. TypeScript

`Vue3` 是基于 `typeScript` 编写的，提供了更好的类型检查，能支持复杂的类型推导

2.2.2. 性能

vue3 是从哪些方面对性能进行进一步优化呢？

- 体积优化
- 编译优化
- 数据劫持优化

这里讲述数据劫持：

在 vue2 中，数据劫持是通过 `Object.defineProperty`，这个 API 有一些缺陷，并不能检测对象属性的添加和删除

```
1 Object.defineProperty(data, 'a', {
2   get(){
3     // track
4   },
5   set(){
6     // trigger
7   }
8 })
```

尽管 Vue 为了解决这个问题提供了 `set` 和 `delete` 实例方法，但是对于用户来说，还是增加了一定的心智负担

同时在面对嵌套层级比较深的情况下，就存在性能问题

```
1 default {
2   data: {
3     a: {
4       b: {
5         c: {
6           d: 1
7         }
8       }
9     }
10  }
11 }
```

相比之下，vue3 是通过 proxy 监听整个对象，那么对于删除还是监听当然也能监听到

同时 Proxy 并不能监听到内部深层次的对象变化，而 Vue3 的处理方式是在 getter 中去递归响应式，这样的好处是真正访问到的内部对象才会变成响应式，而不是无脑递归

2.2.3. 语法 API

这里当然说的就是 `composition API`，其两大显著的优化：

- 优化逻辑组织
- 优化逻辑复用

2.2.3.1. 逻辑组织

一张图，我们可以很直观地感受到 `Composition API` 在逻辑组织方面的优势



相同功能的代码编写在一块，而不像 `options API` 那样，各个功能的代码混成一块

2.2.3.2. 逻辑复用

在 `vue2` 中，我们是通过 `mixin` 实现功能混合，如果多个 `mixin` 混合，会存在两个非常明显的问题：命名冲突和数据来源不清晰

而通过 `composition` 这种形式，可以将一些复用的代码抽离出来作为一个函数，只要使用的地方直接进行调用即可

同样是上文的获取鼠标位置的例子

JavaScript 复制代码

```
1 import { toRefs, reactive, onUnmounted, onMounted } from 'vue';
2 function useMouse(){
3   const state = reactive({x:0,y:0});
4   const update = e=>{
5     state.x = e.pageX;
6     state.y = e.pageY;
7   }
8   onMounted(()=>{
9     window.addEventListener('mousemove',update);
10  })
11  onUnmounted(()=>{
12    window.removeEventListener('mousemove',update);
13  })
14
15  return toRefs(state);
16 }
```

组件使用

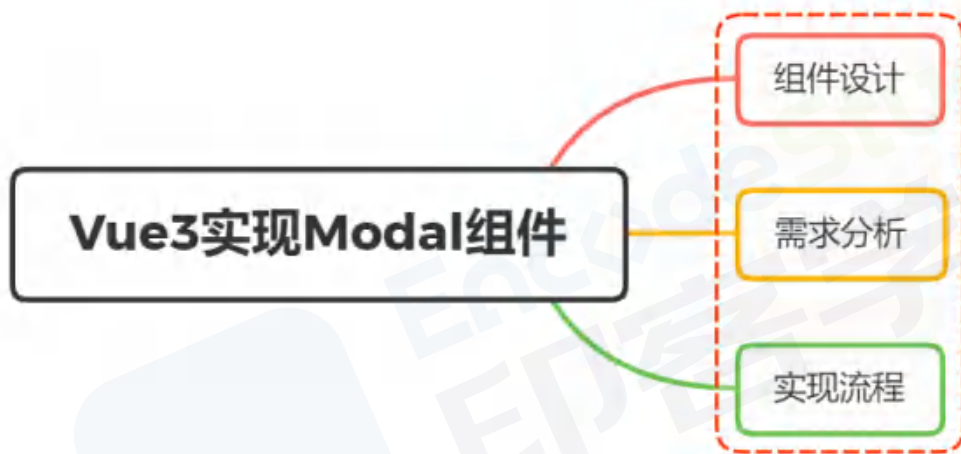
JavaScript 复制代码

```
1 import useMousePosition from './mouse'
2 export default {
3   setup() {
4     const { x, y } = useMousePosition()
5     return { x, y }
6   }
7 }
```

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

3. 用Vue3.0 写过组件吗？如果想实现一个 Modal你会

怎么设计？



3.1. 一、组件设计

组件就是把图形、非图形的各种逻辑均抽象为一个统一的概念（组件）来实现开发的模式

现在有一个场景，点击新增与编辑都弹框出来进行填写，功能上大同小异，可能只是标题内容或者是显示的主体内容稍微不同

这时候就没必要写两个组件，只需要根据传入的参数不同，组件显示不同内容即可

这样，下次开发相同界面程序时就可以写更少的代码，意味着更高的开发效率，更少的 `Bug` 和更少的程序体积

3.2. 需求分析

实现一个 `Modal` 组件，首先确定需要完成的内容：

- 遮罩层
- 标题内容
- 主体内容
- 确定和取消按钮

主体内容需要灵活，所以可以是字符串，也可以是一段 `html` 代码

特点是它们在当前 `vue` 实例之外独立存在，通常挂载于 `body` 之上

除了通过引入 `import` 的形式，我们还可通过 `API` 的形式进行组件的调用

还可以包括配置全局样式、国际化、与 `typeScript` 结合

3.3. 实现流程

首先看看大致流程：

- 目录结构
- 组件内容
- 实现 API 形式
- 事件处理
- 其他完善

3.3.1. 目录结构

`Modal` 组件相关的目录结构

```
Plain Text | 复制代码

1  | plugins
2  |   └─ modal
3  |       └─ Content.tsx // 维护 Modal 的内容, 用于 h 函数和 jsx 语法
4  |       └─ Modal.vue // 基础组件
5  |       └─ config.ts // 全局默认配置
6  |       └─ index.ts // 入口
7  |       └─ locale // 国际化相关
8  |           └─ index.ts
9  |           └─ lang
10 |               └─ en-US.ts
11 |               └─ zh-CN.ts
12 |               └─ zh-TW.ts
13 |       └─ modal.type.ts // ts类型声明相关
```

因为 `Modal` 会被 `app.use(Modal)` 调用作为一个插件，所以都放在 `plugins` 目录下

3.3.2. 组件内容

首先实现 `modal.vue` 的主体显示内容大致如下

```

1 <Teleport to="body" :disabled="!isTeleport">
2   <div v-if="modelValue" class="modal">
3     <div
4       class="mask"
5       :style="style"
6       @click="maskClose && !loading && handleCancel()"
7     ></div>
8     <div class="modal__main">
9       <div class="modal__title line line--b">
10        <span>{{ title || t("r.title") }}</span>
11        <span
12          v-if="close"
13          :title="t('r.close')"
14          class="close"
15          @click="!loading && handleCancel()"
16        >x</span>
17      >
18    </div>
19    <div class="modal__content">
20      <Content v-if="typeof content === 'function'" :render="con
tent" />
21      <slot v-else>
22        {{ content }}
23      </slot>
24    </div>
25    <div class="modal__btns line line--t">
26      <button :disabled="loading" @click="handleConfirm">
27        <span class="loading" v-if="loading"> ○ </span>{{ t
("r.confirm") }}
28      </button>
29      <button @click="!loading && handleCancel()">
30        {{ t("r.cancel") }}
31      </button>
32    </div>
33  </div>
34 </div>
35 </Teleport>

```

最外层上通过Vue3 `Teleport` 内置组件进行包裹，其相当于传送门，将里面的内容传送至 `body` 之上

并且从 `DOM` 结构上来看，把 `modal` 该有的内容（遮罩层、标题、内容、底部按钮）都实现了关于主体内容

HTML | 复制代码

```
1 <div class="modal__content">
2   <Content v-if="typeof content==='function'"
3     :render="content" />
4   <slot v-else>
5     {{content}}
6   </slot>
7 </div>
```

可以看到根据传入 `content` 的类型不同，对应显示不同得到内容

最常见的则是通过调用字符串和默认插槽的形式

HTML | 复制代码

```
1 // 默认插槽
2 <Modal v-model="show"
3   title="演示 slot">
4   <div>hello world~</div>
5 </Modal>
6
7 // 字符串
8 <Modal v-model="show"
9   title="演示 content"
10  content="hello world~" />
```

通过 API 形式调用 `Modal` 组件的时候，`content` 可以使用下面两种

- h 函数

JavaScript | 复制代码

```
1 $modal.show({
2   title: '演示 h 函数',
3   content(h) {
4     return h(
5       'div',
6       {
7         style: 'color:red;',
8         onClick: ($event: Event) => console.log('clicked', $event.target)
9       },
10      'hello world ~'
11     );
12   }
13 });
```

- JSX

JavaScript | 复制代码

```
1 $modal.show({
2   title: '演示 jsx 语法',
3   content() {
4     return (
5       <div
6         onClick=({$event: Event}) => console.log('clicked', $event.target)}
7       >
8         hello world ~
9       </div>
10    );
11  }
12 });
```

3.3.3. 实现 API 形式

那么组件如何实现 API 形式调用 Modal 组件呢？

在 Vue2 中，我们可以借助 Vue 实例以及 Vue.extend 的方式获得组件实例，然后挂载到 body 上

JavaScript | 复制代码

```
1 import Modal from './Modal.vue';
2 const ComponentClass = Vue.extend(Modal);
3 const instance = new ComponentClass({ el: document.createElement("div") });
4 document.body.appendChild(instance.$el);
```

虽然 Vue3 移除了 Vue.extend 方法，但可以通过 createVNode 实现

JavaScript | 复制代码

```
1 import Modal from './Modal.vue';
2 const container = document.createElement('div');
3 const vnode = createVNode(Modal);
4 render(vnode, container);
5 const instance = vnode.component;
6 document.body.appendChild(container);
```

在 Vue2 中，可以通过 this 的形式调用全局 API

JavaScript | 复制代码

```
1 export default {  
2   install(vue) {  
3     vue.prototype.$create = create  
4   }  
5 }
```

而在 Vue3 的 `setup` 中已经没有 `this` 概念了，需要调用 `app.config.globalProperties` 挂载到全局

JavaScript | 复制代码

```
1 export default {  
2   install(app) {  
3     app.config.globalProperties.$create = create  
4   }  
5 }
```

3.3.4. 事件处理

下面再看看 `Modal` 组件内部是如何处理「确定」「取消」事件的，既然是 `Vue3`，当然采用 `Composition API` 形式

JavaScript | 复制代码

```

1 // Modal.vue
2 setup(props, ctx) {
3   let instance = getCurrentInstance(); // 获得当前组件实例
4   onBeforeMount(() => {
5     instance._hub = {
6       'on-cancel': () => {},
7       'on-confirm': () => {}
8     };
9   });
10
11   const handleConfirm = () => {
12     ctx.emit('on-confirm');
13     instance._hub['on-confirm']();
14   };
15   const handleCancel = () => {
16     ctx.emit('on-cancel');
17     ctx.emit('update:modelValue', false);
18     instance._hub['on-cancel']();
19   };
20
21   return {
22     handleConfirm,
23     handleCancel
24   };
25 }

```

在上面代码中，可以看得到除了使用传统 `emit` 的形式使父组件监听，还可通过 `_hub` 属性中添加 `on-cancel`，`on-confirm` 方法实现在 API 中进行监听

JavaScript | 复制代码

```

1 app.config.globalProperties.$modal = {
2   show({}) {
3     /* 监听 确定、取消 事件 */
4   }
5 }

```

下面再来目睹下 `_hub` 是如何实现


```
1 // index.ts
2 app.config.globalProperties.$modal = {
3   show({
4     /* 其他选项 */
5     onConfirm,
6     onCancel
7   }) {
8     /* ... */
9
10    const { props, _hub } = instance;
11
12    const _closeModal = () => {
13      props.modelValue = false;
14      container.parentNode!.removeChild(container);
15    };
16    // 往 _hub 新增事件的具体实现
17    Object.assign(_hub, {
18      async 'on-confirm'() {
19        if (onConfirm) {
20          const fn = onConfirm();
21          // 当方法返回为 Promise
22          if (fn && fn.then) {
23            try {
24              props.loading = true;
25              await fn;
26              props.loading = false;
27              _closeModal();
28            } catch (err) {
29              // 发生错误时，不关闭弹框
30              console.error(err);
31              props.loading = false;
32            }
33          } else {
34            _closeModal();
35          }
36        } else {
37          _closeModal();
38        }
39      },
40      'on-cancel'() {
41        onCancel && onCancel();
42        _closeModal();
43      }
44    });
45  }
```

3.3.5. 其他完善

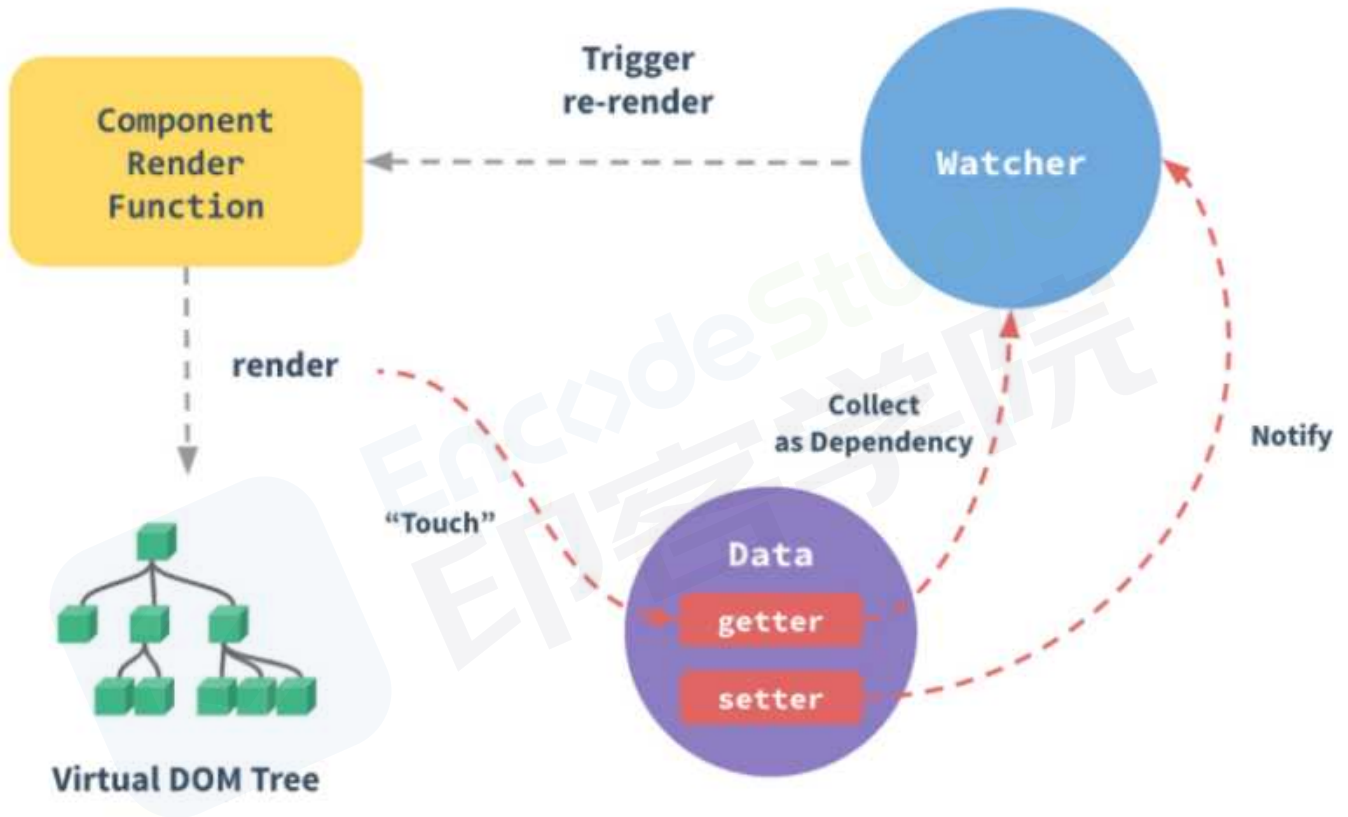
关于组件实现国际化、与 `typescript` 结合，大家可以根据自身情况在此基础上进行更改

4. Vue3.0性能提升主要是通过哪几方面体现的?



4.1. 编译阶段

回顾 `Vue2`，我们知道每个组件实例都对应一个 `watcher` 实例，它会在组件渲染的过程中把用到的数据 `property` 记录为依赖，当依赖发生改变，触发 `setter`，则会通知 `watcher`，从而使关联的组件重新渲染



试想一下，一个组件结构如下图

```
HTML | 复制代码
1 <template>
2   <div id="content">
3     <p class="text">静态文本</p>
4     <p class="text">静态文本</p>
5     <p class="text">{{ message }}</p>
6     <p class="text">静态文本</p>
7     ...
8     <p class="text">静态文本</p>
9   </div>
10 </template>
```

可以看到，组件内部只有一个动态节点，剩余一堆都是静态节点，所以这里很多 `diff` 和遍历其实都是不需要的，造成性能浪费

因此，`Vue3` 在编译阶段，做了进一步优化。主要有如下：

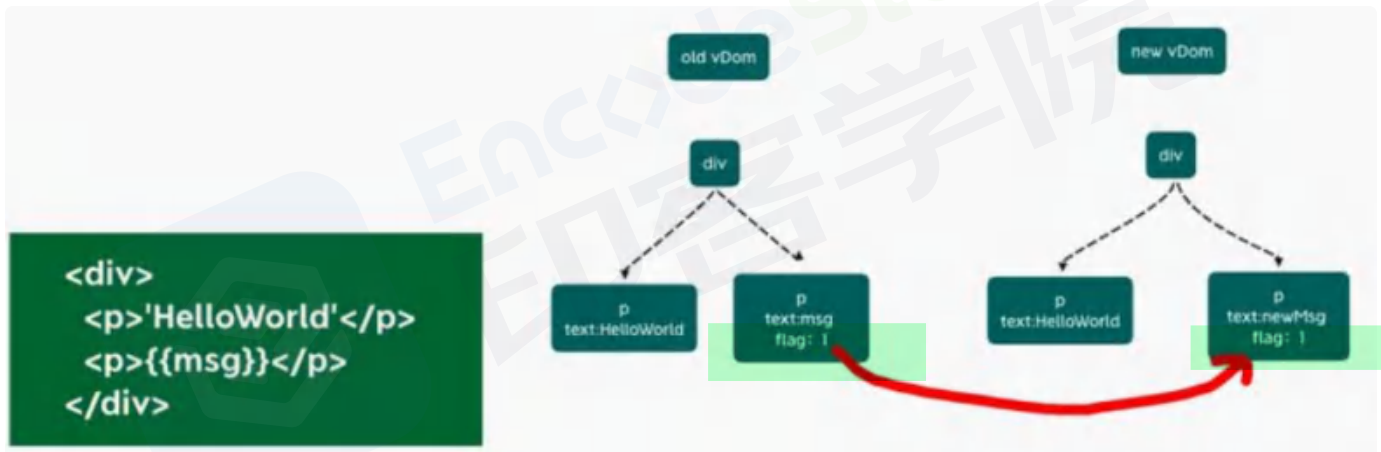
- `diff`算法优化
- 静态提升
- 事件监听缓存
- SSR优化

4.1.1.1. diff算法优化

vue3 在 diff 算法中相比 vue2 增加了静态标记

关于这个静态标记，其作用是为了会发生变化的地方添加一个 flag 标记，下次发生变化的时候直接找该地方进行比较

下图这里，已经标记静态节点的 p 标签在 diff 过程中则不会比较，把性能进一步提高



关于静态类型枚举如下

JavaScript | 复制代码

```
1 export const enum PatchFlags {
2   TEXT = 1, // 动态的文本节点
3   CLASS = 1 << 1, // 2 动态的 class
4   STYLE = 1 << 2, // 4 动态的 style
5   PROPS = 1 << 3, // 8 动态属性，不包括类名和样式
6   FULL_PROPS = 1 << 4, // 16 动态 key，当 key 变化时需要完整的 diff 算法做比较
7   HYDRATE_EVENTS = 1 << 5, // 32 表示带有事件监听器的节点
8   STABLE_FRAGMENT = 1 << 6, // 64 一个不会改变子节点顺序的 Fragment
9   KEYED_FRAGMENT = 1 << 7, // 128 带有 key 属性的 Fragment
10  UNKEYED_FRAGMENT = 1 << 8, // 256 子节点没有 key 的 Fragment
11  NEED_PATCH = 1 << 9, // 512
12  DYNAMIC_SLOTS = 1 << 10, // 动态 slot
13  HOISTED = -1, // 特殊标志是负整数表示永远不会用作 diff
14  BAIL = -2 // 一个特殊的标志，指代差异算法
15 }
```

4.1.1.2. 静态提升

Vue3 中对不参与更新的元素，会做静态提升，只会被创建一次，在渲染时直接复用

这样就免去了重复的创建节点，大型应用会受益于这个改动，免去了重复的创建操作，优化了运行时候的内存占用

JavaScript | 复制代码

```

1 <span>你好</span>
2
3 <div>{{ message }}</div>

```

没有做静态提升之前

JavaScript | 复制代码

```

1 export function render(_ctx, _cache, $props, $setup, $data, $options) {
2   return (_openBlock(), _createBlock(_Fragment, null, [
3     _createVNode("span", null, "你好"),
4     _createVNode("div", null, _toDisplayString(_ctx.message), 1 /* TEXT */)
5   ], 64 /* STABLE_FRAGMENT */))
6 }

```

做了静态提升之后

JavaScript | 复制代码

```

1 const _hoisted_1 = /*#__PURE__*/_createVNode("span", null, "你好", -1 /* HOISTED */)
2
3 export function render(_ctx, _cache, $props, $setup, $data, $options) {
4   return (_openBlock(), _createBlock(_Fragment, null, [
5     _hoisted_1,
6     _createVNode("div", null, _toDisplayString(_ctx.message), 1 /* TEXT */)
7   ], 64 /* STABLE_FRAGMENT */))
8 }
9
10 // Check the console for the AST

```

静态内容 `_hoisted_1` 被放置在 `render` 函数外，每次渲染的时候只要取 `_hoisted_1` 即可
同时 `_hoisted_1` 被打上了 `PatchFlag`，静态标记值为 `-1`，特殊标志是负整数表示永远不会用于 Diff

4.1.1.3. 事件监听缓存

默认情况下绑定事件行为会被视为动态绑定，所以每次都会去追踪它的变化

LaTeX | 复制代码

```

1 <div>
2   <button @click = 'onClick'>点我</button>
3 </div>

```

没开启事件监听器缓存

JavaScript | 复制代码

```

1 export const render = /*#__PURE__*/_withId(function render(_ctx, _cache, $p
  props, $setup, $data, $options) {
2   return (_openBlock(), _createBlock("div", null, [
3     _createVNode("button", { onClick: _ctx.onClick }, "点我", 8 /* PROPS */
    , ["onClick"]))
4                                     // PROPS=1<<3, // 8 //动态属性,
    但不包含类名和样式
5   ]))
6 })

```

开启事件侦听器缓存后

JavaScript | 复制代码

```

1 export function render(_ctx, _cache, $props, $setup, $data, $options) {
2   return (_openBlock(), _createBlock("div", null, [
3     _createVNode("button", {
4       onClick: _cache[1] || (_cache[1] = (...args) => (_ctx.onClick(...args
    )))
5     }, "点我")
6   ]))
7 }

```

上述发现开启了缓存后，没有了静态标记。也就是说下次 diff 算法的时候直接使用

4.1.1.4. SSR优化

当静态内容大到一定量级时候，会用 `createStaticVNode` 方法在客户端去生成一个static node，这些静态 node，会被直接 `innerHTML`，就不需要创建对象，然后根据对象渲染

JavaScript | 复制代码

```

1  div>
2    <div>
3      <span>你好</span>
4    </div>
5    ... // 很多个静态属性
6    <div>
7      <span>{{ message }}</span>
8    </div>
9  </div>

```

编译后

JavaScript | 复制代码

```

1  import { mergeProps as _mergeProps } from "vue"
2  import { ssrRenderAttrs as _ssrRenderAttrs, ssrInterpolate as _ssrInterpolate } from "@vue/server-renderer"
3
4  export function ssrRender(_ctx, _push, _parent, _attrs, $props, $setup, $data, $options) {
5    const _cssVars = { style: { color: _ctx.color }}
6    _push(`<div${
7      _ssrRenderAttrs(_mergeProps(_attrs, _cssVars))
8    }><div><span>你好</span>...<div><span>你好</span><div><span>${
9      _ssrInterpolate(_ctx.message)
10    }</span></div></div>`)
11  }

```

4.2. 源码体积

相比 `Vue2`，`Vue3` 整体体积变小了，除了移出一些不常用的API，再重要的是 `Tree shanking`

任何一个函数，如 `ref`、`reactived`、`computed` 等，仅仅在用到的时候才打包，没用到的模块都被摇掉，打包的整体体积变小

```
1  import { computed, defineComponent, ref } from 'vue';
2  export default defineComponent({
3    setup(props, context) {
4      const age = ref(18)
5
6      let state = reactive({
7        name: 'test'
8      })
9
10     const readOnlyAge = computed(() => age.value++) // 19
11
12     return {
13       age,
14       state,
15       readOnlyAge
16     }
17   }
18 });
```

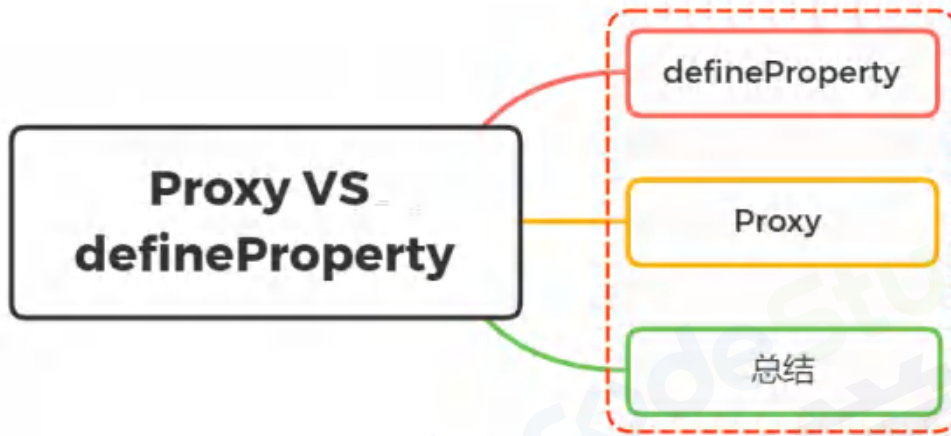
4.3. 响应式系统

vue2 中采用 `defineProperty` 来劫持整个对象，然后进行深度遍历所有属性，给每个属性添加 `getter` 和 `setter`，实现响应式

vue3 采用 `proxy` 重写了响应式系统，因为 `proxy` 可以对整个对象进行监听，所以不需要深度遍历

- 可以监听动态属性的添加
- 可以监听到数组的索引和数组 `length` 属性
- 可以监听删除属性

5. Vue3.0里为什么要用 Proxy API 替代 defineProperty API ?



5.1. Object.defineProperty

定义：`Object.defineProperty()` 方法会直接在一个对象上定义一个新属性，或者修改一个对象的现有属性，并返回此对象

5.1.1. 为什么能实现响应式

通过 `defineProperty` 两个属性，`get` 及 `set`

- `get`

属性的 getter 函数，当访问该属性时，会调用此函数。执行时不传入任何参数，但是会传入 `this` 对象（由于继承关系，这里的 `this` 并不一定是定义该属性的对象）。该函数的返回值会被用作属性的值

- `set`

属性的 setter 函数，当属性值被修改时，会调用此函数。该方法接受一个参数（也就是被赋予的新值），会传入赋值时的 `this` 对象。默认为 `undefined`

下面通过代码展示：

定义一个响应式函数 `defineReactive`

JavaScript | 复制代码

```

1 function update() {
2     app.innerText = obj.foo
3 }
4
5 function defineReactive(obj, key, val) {
6     Object.defineProperty(obj, key, {
7         get() {
8             console.log(`get ${key}:${val}`);
9             return val
10        },
11        set(newVal) {
12            if (newVal !== val) {
13                val = newVal
14                update()
15            }
16        }
17    })
18 }

```

调用 `defineReactive`，数据发生变化触发 `update` 方法，实现数据响应式

JavaScript | 复制代码

```

1 const obj = {}
2 defineReactive(obj, 'foo', '')
3 setTimeout(()=>{
4     obj.foo = new Date().toLocaleTimeString()
5 },1000)

```

在对象存在多个 `key` 情况下，需要进行遍历

JavaScript | 复制代码

```

1 function observe(obj) {
2     if (typeof obj !== 'object' || obj == null) {
3         return
4     }
5     Object.keys(obj).forEach(key => {
6         defineReactive(obj, key, obj[key])
7     })
8 }

```

如果存在嵌套对象的情况，还需要在 `defineReactive` 中进行递归

JavaScript | 复制代码

```

1 function defineReactive(obj, key, val) {
2   observe(val)
3   Object.defineProperty(obj, key, {
4     get() {
5       console.log(`get ${key}:${val}`);
6       return val
7     },
8     set(newVal) {
9       if (newVal !== val) {
10        val = newVal
11        update()
12      }
13    }
14  })
15 }

```

当给 `key` 赋值为对象的时候，还需要在 `set` 属性中进行递归

JavaScript | 复制代码

```

1 set(newVal) {
2   if (newVal !== val) {
3     observe(newVal) // 新值是对象的情况
4     notifyUpdate()
5   }
6 }

```

上述例子能够实现对一个对象的基本响应式，但仍然存在诸多问题

现在对一个对象进行删除与添加属性操作，无法劫持到

JavaScript | 复制代码

```

1 const obj = {
2   foo: "foo",
3   bar: "bar"
4 }
5 observe(obj)
6 delete obj.foo // no ok
7 obj.jar = 'xxx' // no ok

```

当我们对一个数组进行监听的时候，并不那么好使了

```
1  const arrData = [1,2,3,4,5];
2  arrData.forEach((val, index)=>{
3      defineProperty(arrData, index, val)
4  })
5  arrData.push() // no ok
6  arrData.pop()  // no ok
7  arrDate[0] = 99 // ok
```

可以看到数据的 `api` 无法劫持到，从而无法实现数据响应式，

所以在 `Vue2` 中，增加了 `set`、`delete` API，并且对数组 `api` 方法进行一个重写

还有一个问题则是，如果存在深层的嵌套对象关系，需要深层的进行监听，造成了性能的极大问题

5.1.2. 小结

- 检测不到对象属性的添加和删除
- 数组 `API` 方法无法监听到
- 需要对每个属性进行遍历监听，如果嵌套对象，需要深层监听，造成性能问题

5.2. proxy

`Proxy` 的监听是针对一个对象的，那么对这个对象的所有操作会进入监听操作，这就完全可以代理所有属性了

在 `ES6` 系列中，我们详细讲解过 `Proxy` 的使用，就不再赘述了

下面通过代码进行展示：

定义一个响应式方法 `reactive`

JavaScript | 复制代码

```
1 function reactive(obj) {
2   if (typeof obj !== 'object' && obj !== null) {
3     return obj
4   }
5   // Proxy相当于在对象外层加拦截
6   const observed = new Proxy(obj, {
7     get(target, key, receiver) {
8       const res = Reflect.get(target, key, receiver)
9       console.log(`获取${key}:${res}`)
10      return res
11    },
12    set(target, key, value, receiver) {
13      const res = Reflect.set(target, key, value, receiver)
14      console.log(`设置${key}:${value}`)
15      return res
16    },
17    deleteProperty(target, key) {
18      const res = Reflect.deleteProperty(target, key)
19      console.log(`删除${key}:${res}`)
20      return res
21    }
22  })
23   return observed
24 }
```

测试一下简单数据的操作，发现都能劫持

JavaScript | 复制代码

```
1 const state = reactive({
2   foo: 'foo'
3 })
4 // 1. 获取
5 state.foo // ok
6 // 2. 设置已存在属性
7 state.foo = 'fooooooo' // ok
8 // 3. 设置不存在属性
9 state.dong = 'dong' // ok
10 // 4. 删除属性
11 delete state.dong // ok
```

再测试嵌套对象情况，这时候发现就不那么 OK 了

JavaScript | 复制代码

```

1 const state = reactive({
2   bar: { a: 1 }
3 })
4
5 // 设置嵌套对象属性
6 state.bar.a = 10 // no ok

```

如果要解决，需要在 `get` 之上再进行一层代理

JavaScript | 复制代码

```

1 function reactive(obj) {
2   if (typeof obj !== 'object' && obj !== null) {
3     return obj
4   }
5   // Proxy相当于在对象外层加拦截
6   const observed = new Proxy(obj, {
7     get(target, key, receiver) {
8       const res = Reflect.get(target, key, receiver)
9       console.log(`获取${key}:${res}`)
10      return isObject(res) ? reactive(res) : res
11    },
12    return observed
13  }

```

5.3. 总结

`Object.defineProperty` 只能遍历对象属性进行劫持

JavaScript | 复制代码

```

1 function observe(obj) {
2   if (typeof obj !== 'object' || obj == null) {
3     return
4   }
5   Object.keys(obj).forEach(key => {
6     defineReactive(obj, key, obj[key])
7   })
8 }

```

`Proxy` 直接可以劫持整个对象，并返回一个新对象，我们可以只操作新的对象达到响应式目的

JavaScript | 复制代码

```

1 function reactive(obj) {
2   if (typeof obj !== 'object' && obj !== null) {
3     return obj
4   }
5   // Proxy相当于在对象外层加拦截
6   const observed = new Proxy(obj, {
7     get(target, key, receiver) {
8       const res = Reflect.get(target, key, receiver)
9       console.log(`获取${key}:${res}`)
10      return res
11    },
12    set(target, key, value, receiver) {
13      const res = Reflect.set(target, key, value, receiver)
14      console.log(`设置${key}:${value}`)
15      return res
16    },
17    deleteProperty(target, key) {
18      const res = Reflect.deleteProperty(target, key)
19      console.log(`删除${key}:${res}`)
20      return res
21    }
22  })
23   return observed
24 }

```

Proxy 可以直接监听数组的变化 (push 、 shift 、 splice)

JavaScript | 复制代码

```

1 const obj = [1,2,3]
2 const proxObj = reactive(obj)
3 obj.push(4) // ok

```

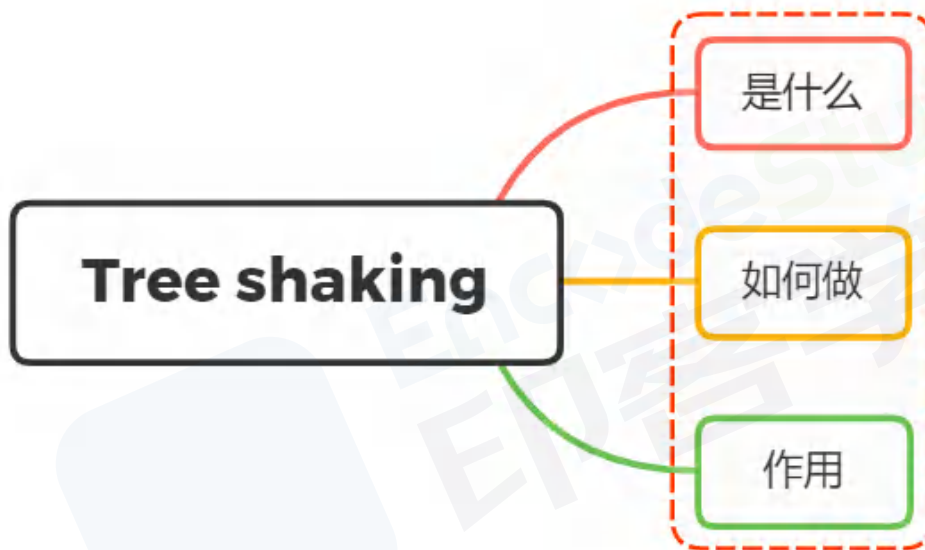
Proxy 有多达13种拦截方法,不限于 apply 、 ownKeys 、 deleteProperty 、 has 等等, 这是 Object.defineProperty 不具备的

正因为 defineProperty 自身的缺陷, 导致 Vue2 在实现响应式过程需要实现其他的方法辅助 (如 重写数组方法、增加额外 set 、 delete 方法)

```
1 // 数组重写
2 const originalProto = Array.prototype
3 const arrayProto = Object.create(originalProto)
4 ['push', 'pop', 'shift', 'unshift', 'splice', 'reverse', 'sort'].forEach(m
  ethod => {
5   arrayProto[method] = function () {
6     originalProto[method].apply(this, arguments)
7     dep.notice()
8   }
9 });
10
11 // set、delete
12 Vue.set(obj, 'bar', 'newbar')
13 Vue.delete(obj, 'bar')
```

Proxy 不兼容IE，也没有 polyfill，defineProperty 能支持到IE9

6. 说说Vue 3.0中Treeshaking特性？举例说明一下？



6.1. 是什么

Tree shaking 是一种通过清除多余代码方式来优化项目打包体积的技术，专业术语叫 Dead code elimination

简单来讲，就是在保持代码运行结果不变的前提下，去除无用的代码

如果把代码打包比作制作蛋糕，传统的方式是把鸡蛋（带壳）全部丢进去搅拌，然后放入烤箱，最后把（没有用的）蛋壳全部挑选并剔除出去

而 `treeshaking` 则是一开始就把有用的蛋白蛋黄（`import`）放入搅拌，最后直接作出蛋糕

也就是说，`tree shaking` 其实是找出使用的代码

在 `Vue2` 中，无论我们使用什么功能，它们最终都会出现在生产代码中。主要原因是 `Vue` 实例在项目

中是单例的，捆绑程序无法检测到该对象的哪些属性在代码中被使用到

```
JavaScript | 复制代码
1  import Vue from 'vue'
2
3  Vue.nextTick(() => {})
```

而 `Vue3` 源码引入 `tree shaking` 特性，将全局 API 进行分块。如果您不使用其某些功能，它们将不会包含在您的基础包中

```
JavaScript | 复制代码
1  import { nextTick, observable } from 'vue'
2
3  nextTick(() => {})
```

6.2. 如何做

`Tree shaking` 是基于 `ES6` 模板语法（`import` 与 `exports`），主要是借助 `ES6` 模块的静态编译思想，在编译时就能确定模块的依赖关系，以及输入和输出的变量

`Tree shaking` 无非就是做了两件事：

- 编译阶段利用 `ES6 Module` 判断哪些模块已经加载
- 判断那些模块和变量未被使用或者引用，进而删除对应代码

下面就来举个例子：

通过脚手架 `vue-cli` 安装 `Vue2` 与 `Vue3` 项目

```
JavaScript | 复制代码
1  vue create vue-demo
```

6.2.1. Vue2 项目

组件中使用 `data` 属性

Vue | 复制代码

```
1 <script>
2   export default {
3     data: () => ({
4       count: 1,
5     }),
6   };
7 </script>
```

对项目进行打包，体积如下图

DONE Compiled successfully in 18393ms

File	Size	Gzipped
dist\js\chunk-vendors.28d0d835.js	89.59 KiB	32.11 KiB
dist\js\app.3f482fc1.js	2.01 KiB	1.01 KiB

Images and other types of assets omitted.

为组件设置其他属性（`computed`、`watch`）

JavaScript | 复制代码

```
1 export default {
2   data: () => ({
3     question:"",
4     count: 1,
5   }),
6   computed: {
7     double: function () {
8       return this.count * 2;
9     },
10  },
11  watch: {
12    question: function (newQuestion, oldQuestion) {
13      this.answer = 'xxx'
14    }
15  };
16 }
```

再一次打包，发现打包出来的体积并没有变化

File	Size	Gzipped
dist\js\chunk-vendors.28d0d835.js	89.59 KiB	32.11 KiB
dist\js\app.94092e3d.js	2.07 KiB	1.04 KiB

Images and other types of assets omitted.

6.2.2. Vue3 项目

组件中简单使用

JavaScript | 复制代码

```
1  import { reactive, defineComponent } from "vue";
2  export default defineComponent({
3    setup() {
4      const state = reactive({
5        count: 1,
6      });
7      return {
8        state,
9      };
10    },
11  });
```

将项目进行打包

DONE Compiled successfully in 12015ms

File	Size	Gzipped
dist\js\chunk-vendors.de2030ce.js	78.91 KiB	29.62 KiB
dist\js\app.c06bf53e.js	1.92 KiB	0.93 KiB

在组件中引入 `computed` 和 `watch`

```

1  import { reactive, defineComponent, computed, watch } from "vue";
2  export default defineComponent({
3    setup() {
4      const state = reactive({
5        count: 1,
6      });
7      const double = computed(() => {
8        return state.count * 2;
9      });
10
11     watch(
12       () => state.count,
13       (count, preCount) => {
14         console.log(count);
15         console.log(preCount);
16       }
17     );
18     return {
19       state,
20       double,
21     };
22   },
23 });

```

再次对项目进行打包，可以看到在引入 `computer` 和 `watch` 之后，项目整体体积变大了

DONE Compiled successfully in 8011ms

File	Size	Gzipped
dist\js\chunk-vendors.19e22567.js	79.05 KiB	29.64 KiB
dist\js\app.00e83bfe.js	2.15 KiB	1.00 KiB

6.3. 作用

通过 `Tree shaking`，`Vue3` 给我们带来的好处是：

- 减少程序体积（更小）
- 减少程序执行时间（更快）
- 便于将来对程序架构进行优化（更友好）