Composition API （组合 API）

这是 Vue3 新特性中最重要的一个部分，如果大家熟悉 React Hook，你也可以叫它 Vue Hook。

动机和目的

简单的描述：**低侵入、函数式 API，更灵活地组合组件逻辑。** 同时，目的中最主要也是面向更加复杂，更大的应用和软件项目考虑。

- 更好的逻辑复用与代码组织

大型项目实践，多人项目实践，长期迭代和维护条件下，如何保持 Vue 项目出现的问题：

1. 概念增长，代码阅读和理解的难度增加
2. 如何更加简洁且低成本的机制提取与重用多个组件之间的逻辑?

- 更好的类型推导

为了更好的支持 Typescript 过程中，原来的体系遇到了不小的麻烦。Vue 原有的设计没有考虑到类型推导的问题，适配 Typescript 非常麻烦！

使用 Typescript class 以及 decorator 模式来支持 Typescript 存在较大风险。

基本范例

我们从一个范例入手，了解组合 API 的基本结构和使用方法

```
<template>
<button @click="increment">
    Count is: {{ state.count }}, double is: {{ state.double }}
</button>
</template>

<script>
import { reactive, computed } from 'vue'

export default {
    setup() {
        const state = reactive({
            count: 0,
            double: computed(() => state.count * 2),
        })


        function increment() {
            state.count++
        }

        return {
```

```
            state,
            increment,
        }
    },
}
</script>
```

整个 API 的引入需要关注这样几个要点

- 响应式状态与副作用
- 计算属性
- 生命周期钩子函数
- `setup()`
- 代码结构
- 复用和逻辑提取

响应式状态和副作用

看一下基本范例中对响应式状态的定义

```
import { reactive, computed } from 'vue'

// state 现在是一个响应式状态
const state = reactive({
    count: 0,
})
```

所谓响应式状态，和我们在 2.x 中利用 data 属性定义的状态是一样的，可以在模版渲染时使用。而在 DOM 中渲染内容会被视为一种"副作用"：程序在外部修改其本身（也就是这个 DOM）的状态。API 提供了`watchEffect`方法应用基于响应式状态的副作用，而且这个方法在内部包含的响应式状态发生变化时自动重新应用。下面就是一个基于前面定义的 state 响应状态的副作用

```
watchEffect(() => {
    document.body.innerHTML = `count is ${state.count}`
})
```

`watchEffect`接收一个应用的预期副作用的函数（上例中就是设置`innerHTML`）。这个方法在代码执行过程中会立刻执行，然后方法中所有响应式状态会被作为依赖继续跟踪，有变化时会再次执行。上例中会在`state.count`变化时执行。

上面就是 Vue 响应式系统的**精髓**所在了！

2.x 中提供的`watch`选项和`watchEffect`效果类似，3.x 也同样支持`watch`，但是它不需要指定依赖的数据源，也不需要将其和副作用分离。

上例中对于响应式状态`state.count`的使用，如模版渲染显示和 2.x 类似，通过上述响应式定义和副作用函数可以专注在处理过程，而无需专门考虑渲染。

计算属性

2.x 中提供了 computed 选项，可以用来定义依赖于其它状态的状态，API 中提供了 computed 方法直接创建一个计算值，参考基本范例中的定义

```
import { reactive, computed } from 'vue'

const state = reactive({

    double: computed(() => state.count * 2),
})
```

`double`就是一个计算属性，它依赖于 state.count 状态。

ref

另外，在 API 中提供了`ref` API 可以直接创建一个可变更的普通参考值，这个值也是响应式状态。

```
import { ref } from "vue";

const count = ref(0);

function increment() {
    count.value ++;
}

const renderContext = {
    count,
    increment,
}
```

上面这段示例代码使用了 ref 来创建响应式状态 count，它可以和前面的基本范例产生相同的效果。

这里注意`ref`对应值类型，而`reactive`对应对象。

`setup`

结合上述信息，在组件当中就可以利用`setup()`将组件和渲染模版组合在一起交给框架处理，这样我们只需要`setup()`函数和模版定义一个组件。这就是我们的基本范例。而且这也是每个熟悉 Vue 的开发人员了解的单文件组件格式。其中发生变化的仅仅是脚本部分的代码。

生命周期钩子函数

同 2.x 中的组件对比，我们已经覆盖了响应式状态、计算属性、用户输入的状态变更，剩下还有就是一些特定的生命周期变化响应，包括状态变化和组件生命周期，API 提供了对应的生命周期响应方法，使用`onXXX`的形式，例如：

```
import { onMounted } from 'vue'

export default {
    setup() {
        onMounted(() => {
            console.log('component is mounted!')
        })
    },
}
```

这里需要注意，生命周期注册方法只能用在`setup`钩子中。

代码组织

通过上面的基本范例，对比 2.x 中使用的选项定义组件的方式，特别是对于复杂的组件，会带来很好组织结构效果。通过逻辑关注点，可以将相同逻辑关注点的代码组织并列在一起，而不会因为选项定义的要求而分离在不同选项中，甚至距离遥远。这样在维护和阅读代码时就不需要上下跳转查看代码，而是可以集中在一起。而且利用这种逻辑关注点，可以将这些逻辑封装在一个函数中，通过使用具有描述性的名称（建议使用`use`开头的函数名），表示一个组合函数。基本范例就可以封装如下

```
import { reactive, computed } from 'vue'

export default function(){
    const state = reactive({
        count: 0,
        double: computed(() => state.count * 2),
    })


    function increment() {
        state.count++
    }

    return {
        state,
        increment,
    }
}
```

这样形成一个可以复用的组合函数组件，利用该组件的演示如下：

```
import useState from "user-state"

export default {
    setup() {
        const { state, increment } = useState();

        return {
            state,
            increment,
        }
    }
}
```

使用这样的组合式 API 提取复用组件逻辑时是十分灵活的。而且组合 API 仅依赖于它的参数和 Vue 全局导出的 API，不再依赖 2.x 中非常微妙的`this`上下文。

通过上述讲解，组合式 API 的好处可以总结一下

- 暴露给模版的属性状态来源清晰，由组件逻辑函数返回
- 不存在命名空间冲突，可以结构为任意命名
- 不再需要仅为逻辑复用而创建的组件实例

同时，组合 API 可以完全与现有的选项 API 搭配使用。

- `setup`在选项之前解析，组合 API 不能提前访问选项中的状态
- `setup()`函数返回的属性会暴露给`this`，在选项中可以访问

其它

`@vue/composition`库将组合 API 以插件形式在 2.x 中生效，因此不需要 3.x 也可以在现有库中使用。但是`import`API 需要从上述插件中获取，不能从`vue`中获取

```
// 2.x
import { ref } from "@vue/composition"

// 3.x
import { ref } from "vue"
```