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>
```

组合 API 的引入有下面几个关注点：

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

setup()

从基本范例中我们首先会发现setup()函数，这个函数是组合 API 引入的组件选项，做为组合 API 的入口点。setup()函数在组件实例创建、初始化 props 之后立刻执行，在生命周期中仅执行一次。

基本范例中的setup()函数返回一个对象，对象中的属性会由 Vue 框架合并到模版的渲染上下文当中。如果和选项设置对应来看，属性中的数据部分对应选项的data定义，而方法对应methods定义。范例中的state相当于数据定义，而increment就是方法定义。

更加详细的关于setup方法的定义可以参考组合 API 的手册，这里不展开说明。

响应式状态和副作用

基本范例中对应响应式状态的定义部分：

```
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 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。

上述就是 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

响应式状态还可以使用 API 提供的`ref` 方法，它可以直接创建一个可变更的普通参考值。如果参数为对象，则调用`reactive`进行深层响应转换

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

const count = ref(0);

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

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。

`ref`创建响应式对象，有一个单一的属性`value`，用于访问值。

对于响应式对象的属性访问，如果`ref`做为 reactive 对象的属性被访问或者修改时，会自动解套 value 值，行为类似于普通属性，参考下面的例子：

```
const count = ref(0)
const state = reactive({
count,
})

console.log(state.count) // 0

state.count = 1
console.log(count.value) // 1
```

但是，如果将一个新的 ref 分配给现有的 ref，将会替换旧的 ref，参考例子：

```
const otherCount = ref(2)

state.count = otherCount
console.log(state.count) // 2
console.log(count.value) // 1
```

注意只有嵌套在 reactive 的响应式对象中，ref 才会解套。如果从其它类似`Array`或者`Map`等原生对象类中访问`ref`，不会自动解套

```
const arr = reactive([ref(0)])
// 这里需要 .value
console.log(arr[0].value)

const map = reactive(new Map([['foo', ref(0)]]))
// 这里需要 .value
console.log(map.get('foo').value)
```

组合 API 提供了更多的响应式系统方法，可以满足各种不同的场景和需求

- readonly
- unref
- toRef
- toRefs
- isRef
- isProxy
- isReactive
- isReadonly
- customRef
- markRaw
- shallowReactive
- shallowReadonly
- shallowRef

- toRaw

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，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，组合 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()`方法中使用。

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，会带来很好的代码组织结构效果。通过将相同逻辑关注点的代码组织并列在一起，不会因为选项定义的要求而分离在不同选项中，使得距离遥远，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。按照这样的方式，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，可以解构为任意命名
- 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"
```