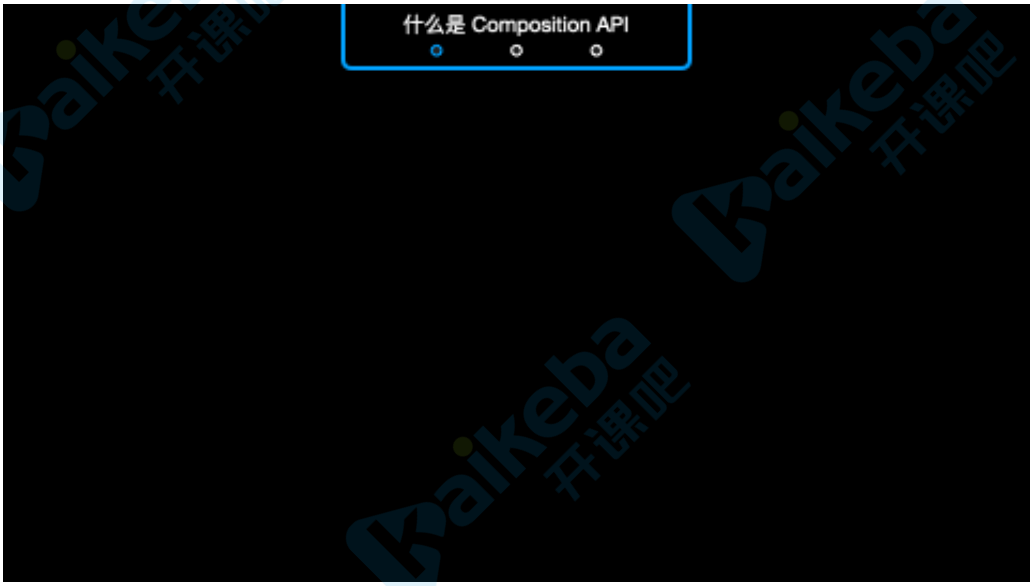
1.3 Composition API

一组低侵入式的、函数式的 API，使得我们能够更灵活地「**组合**」组件的逻辑。

1.3.1 介绍

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，当组件变得更大时，**逻辑关注点**的列表也会增长，这种碎片化使得理解和维护复杂组件变得困难。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。

如果能够**将同一个逻辑关注点相关的代码配置在一起**会更好，这正是 Composition API设计的动机。



1.3.2 setup

`setup` 函数是一个新的组件选项。作为在组件内使用 Composition API 的入口点。

如果 `setup` 返回一个对象，则对象的属性将被合并到组件的渲染函数上下文：

```
<template>
  <div>{{ state.foo }}</div>
</template>

<script>
  import { reactive } from 'vue'

  export default {
    setup() {
      const state = reactive({ foo: 'bar' })

      // 暴露给模板
      return {
```

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

`setup` 也可以返回一个函数，该函数会作为组件渲染函数：

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

export default {
  setup() {
    const state = reactive({ foo: 'bar' })

    // 返回一个函数作为渲染函数
    return () => h('div', state.foo)
  },
}
```

setup 参数：`setup(props, {attrs, slots, emit})`

```
const Comp = {
  template: `<div>comp <slot /></div>`,
  props: {
    dong: {
      type: String,
      default: ''
    },
  },
  setup(props, ctx) {
    console.log(props, ctx);
  }
}
```

```
<comp dong="dong" tua="tua">
  default slot content
</comp>
```

> props是响应式的，但是不能解构，否则将失去响应能力

```
// ok
setup(props) {
watchEffect(() => {
 console.log(props.dong);
})
}
// no ok
setup({dong}) {
watchEffect(() => {
 console.log(dong);
})
}
```

`this` 在 `setup()` 中不可用也没有必要使用

```
setTimeout(() => state.foo = 'barrrrrr', 1000)
```

在`setup()`中获取组件实例

```
const instance = getCurrentInstance()
console.log(instance);
```

范例：课程表范例修改为composition

```
Vue.createApp({
  setup() {
    const state = Vue.reactive({
      courses: [],
      course: '',
      showMsg: false
    })

    setTimeout(() => {
      state.courses = ["web全栈架构师", "web高级工程师"];
    }, 1000);

    function addCourse() {
      state.courses.push(state.course);
      state.course = ''

      state.showMsg = true
    }
```

```
    return { state, addCourse }
  },
})
```

1.3.3 Reactivity API

`reactive`：对象响应式

接收一个普通对象然后返回该普通对象的响应式代理。等同于 vue 2.x 的 `Vue.observable()`

```
const obj = reactive({ count: 0 })
```

`ref`：单值响应式

接受一个参数值并返回一个响应式Ref 对象。Ref 对象拥有一个指向内部值的单一属性 `value`。

```
const count = ref(0)
console.log(count.value) // 0

count.value++
console.log(count.value) // 1
```

> 如果传入 ref 的是一个对象，将调用 `reactive` 方法进行深层响应转换。

模板中访问：Ref对象在模板中使用时会自动解套，无需额外书写 `.value`：

```
<div>{{ count }}</div>
```

范例：改造课程表案例中showMsg为Ref形式

```
const showMsg = Vue.ref(false)

function addCourse() {
  showMsg.value = true
}

return { state, showMsg, addCourse }
```

视图中不再需要state

```
<message v-if="showMsg" @close="showMsg = false">
```

Ref对象作为 reactive 对象的属性被访问或修改时，也将自动解套 value 值：

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

console.log(state.count) // 0

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

范例：改造课程表案例中showMsg为Ref形式，但在reactive中声明

```
const state = Vue.reactive({
  showMsg: Vue.ref(false) // 作为reactive对象属性
})

function addCourse() {
  state.showMsg = true // 可以自动解套value值
}

return { showMsg: state.showMsg } // 以showMsg暴露state.showMsg
```

`toRefs`把一个响应式对象转换成普通对象，该普通对象的每个属性都是一个Ref。

```
const state = reactive({
  foo: 1,
  bar: 2,
})
const stateAsRefs = toRefs(state)
/* stateAsRefs 的类型如下:
{
  foo: Ref<number>,
  bar: Ref<number>
}
*/
```

范例：改造课程表案例，使用toRefs转换state，视图中可避免引用state

```
return { ...Vue.toRefs(state) }
```

```
<course-add v-model:course="course" @add="addCourse"></course-add>
<course-list :courses="courses"></course-list>
```

`computed`：计算属性

传入一个 getter 函数，返回一个**不可手动修改的 Ref 对象** 。

```
const count = ref(1)
const doubleCount = computed(() => count.value * 2)
console.log(doubleCount.value) // 2
doubleCount.value++ // 错误
```

传入一个拥有 `get` 和 `set` 函数的对象，创建一个可手动修改的计算状态。

```
const count = ref(1)
const doubleCount = computed({
  get: () => count.value * 2,
  set: (val) => {
    count.value = val / 2
  },
})

doubleCount.value = 4
console.log(count.value) // 2
```

范例：课程表中显示课程总数

```
<p>课程总数：{{courseCount}}</p>
```

```
return {
  courseCount: Vue.computed(() => state.courses.length + '门'),
}
```

`watchEffect`：副作用侦听器

立即执行传入的一个函数，并收集响应式的依赖，当依赖变更时重新运行该函数。

```
const count = ref(0)

watchEffect(() => console.log(count.value)) // 打印出 0

setTimeout(() => {
  count.value++  // 打印出 1
}, 100)
```

范例：课程总数使用watchEffect实现

```
const state = Vue.reactive({
  courseCount: 0,
})
Vue.watchEffect(() => {
  // state.courses.length访问时被作为依赖，它只要变化就重新设置courseCount
  state.courseCount = state.courses.length + '门'
})
```

`watch`：侦听器

`watch`侦听特定数据源，并在回调函数中执行副作用。

侦听单个数据源：数据源可以是一个拥有返回值的 getter 函数，也可以是 ref：

```
// 侦听一个 getter
const state = reactive({ count: 0 })
watch(
  () => state.count,
  (count, prevCount) => {}
)

// 直接侦听一个 ref
const count = ref(0)
watch(count, (count, prevCount) => {})
```

> watch完全等效于vue2中的this.$watch（包括选项）

范例：缓存数据到localStorage

```
const state = Vue.reactive({
        courses: JSON.parse(localStorage.getItem('courses')) || [],
})
Vue.watch(() => state.courses, () => {
  localStorage.setItem('courses', JSON.stringify(state.courses))
}, {
  deep: true
})
```

侦听多个数据源

`watcher` 也可以使用数组来同时侦听多个源：

```
watch([fooRef, barRef], ([foo, bar], [prevFoo, prevBar]) => {})
```

对比 `watchEffect` 和 `watch` ：

- watch懒执行副作用；
- watch需明确哪些状态改变触发重新执行副作用；
- watch可访问侦听状态变化前后的值。

1.3.4 生命周期钩子

生命周期钩子可以通过 `onXXX` 形式导入并在setup内部注册：

```
import { onMounted, onUpdated, onUnmounted } from 'vue'

const MyComponent = {
  setup() {
    onMounted(() => {
      console.log('mounted!')
    })
    onUpdated(() => {
      console.log('updated!')
    })
    onUnmounted(() => {
      console.log('unmounted!')
    })
  },
}
```

> 注意：这些生命周期钩子注册函数只能在 `setup()` 使用。

可以多次注册，按顺序执行

```
setup() {
  onMounted(() => {
    console.log('mounted1')
  })
  onMounted(() => {
    console.log('mounted2')
  })
}
```

妙用：可以用在其他可复用的逻辑中

```
function useCounter() {
  const counter = ref(0)
  let timer
  onMounted(() => {
    timer = setInterval(() => counter.value++, 1000)
  })
  onUnmounted(() => {
    clearInterval(timer)
  })
  return counter
}

setup() {
  const counter = useCounter()
  return { counter }
}
```

与 2.x 版本生命周期相对应的组合式 API

- `beforeCreate` -> 直接写到 `setup()`
- `created` -> 直接写到 `setup()`
- `beforeMount` -> `onBeforeMount`
- `mounted` -> `onMounted`
- `beforeUpdate` -> `onBeforeUpdate`
- `updated` -> `onUpdated`
- `beforeDestroy` -> `onBeforeUnmount` **变化**
- `destroyed` -> `onUnmounted` **变化**
- `errorCaptured` -> `onErrorCaptured`
- `onRenderTracked` **新增**
- `onRenderTriggered` **新增**

1.3.4 依赖注入

在setup中依赖注入使用`provide`和`inject`。

```
import { provide, inject } from 'vue'

const Ancestor = {
  setup() {
    provide('colorTheme', 'dark')
  },
}

const Descendent = {
  setup() {
    const theme = inject('colorTheme')
    return {
      theme,
    }
  },
}
```

注入值的响应性

如果注入一个响应式对象，则它的状态变化也可以被侦听。

```
// 提供者响应式数据
const themeRef = ref('dark')
provide('colorTheme', themeRef)

// 使用者响应式数据
const theme = inject('colorTheme')

watchEffect(() => {
  console.log(`theme set to: ${theme.value}`)
})
```

1.3.5 模板引用

当使用组合式 API 时，*reactive refs* 和 *template refs* 的概念已经是统一的。为了获得对模板内元素或组件实例的引用，我们可以像往常一样在`setup()`中声明一个 ref 并返回它：

```
<template>
  <div ref="root"></div>
```

```
</template>

<script>
  import { ref, onMounted } from 'vue'

  export default {
    setup() {
      const root = ref(null)

      onMounted(() => {
        // 挂载后, dom会被赋值给root这个ref对象
        console.log(root.value) // <div/>
      })

      return {
        root,
      }
    },
  }
</script>
```