Composition API（二）

王红元 coderwhy

认识readonly

- 我们通过**reactive或者ref可以获取到一个响应式的对象**，但是某些情况下，我们**传入给其他地方（组件）**的这个响应式对象希望**在另外一个地方（组件）被使用**，但是**不能被修改**，这个时候**如何防止这种情况的出现**呢？
  - Vue3为我们提供了readonly的方法；
  - readonly会返回原生对象的只读代理（也就是它依然是一个Proxy，这是一个proxy的set方法被劫持，并且不能对其进行修改）；

- **在开发中常见的readonly方法会传入三个类型的参数：**
  - 类型一：普通对象；
  - 类型二：reactive返回的对象；
  - 类型三：ref的对象；

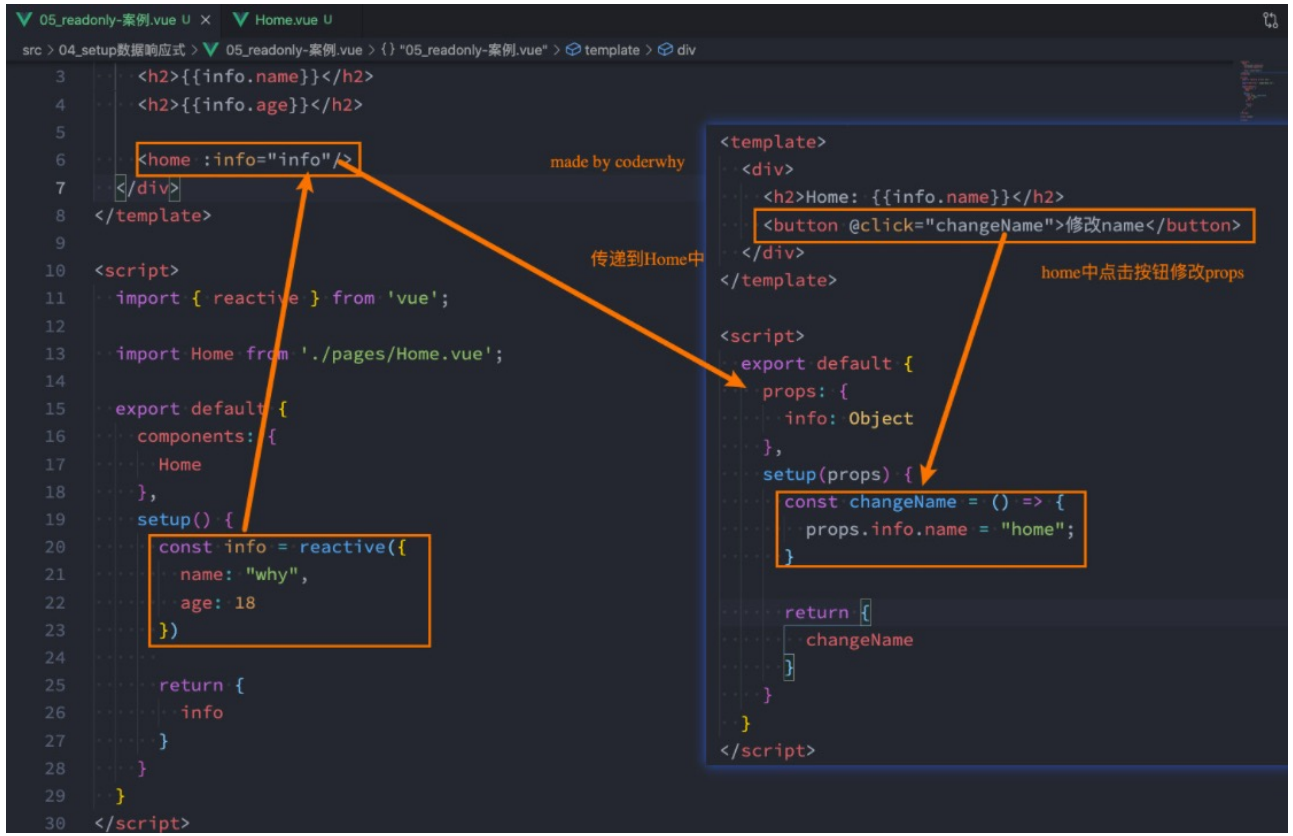
readonly的使用

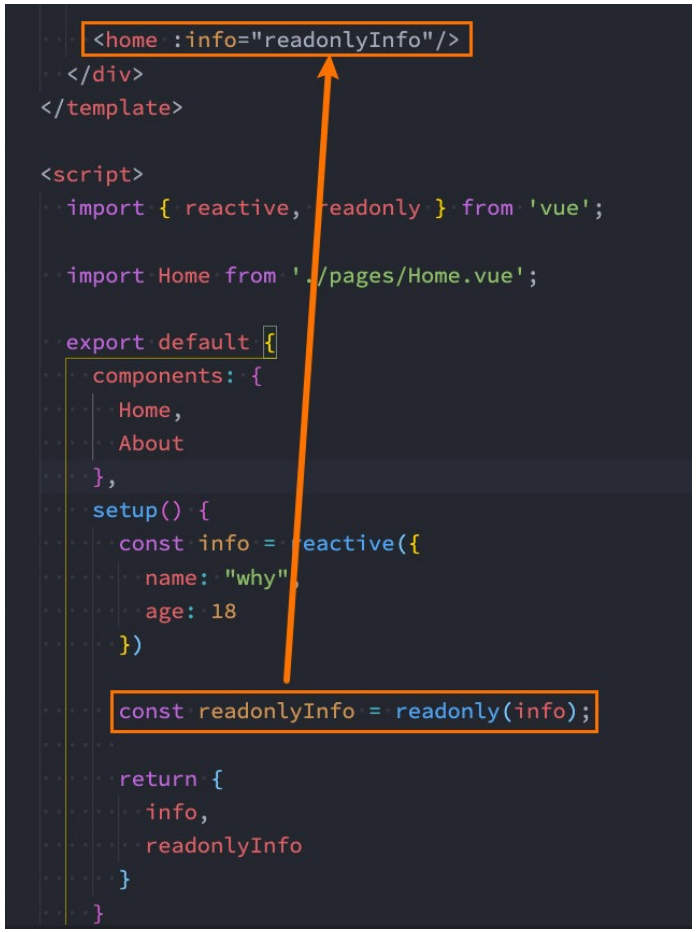
- **在readonly的使用过程中，有如下规则：**
  - readonly返回的对象都是不允许修改的；
  - 但是经过readonly处理的原来的对象是允许被修改的；
    - ✓ 比如 const info = readonly(obj)，info对象是不允许被修改的；
    - ✓ 当obj被修改时，readonly返回的info对象也会被修改；
    - ✓ 但是我们不能去修改readonly返回的对象info；
  - 其实本质上就是readonly返回的对象的setter方法被劫持了而已；

```
// readonly通常会传入三个类型的数据
// 1.传入一个普通对象
const info = {
  name: "why",
  age: 18
}
const state1 = readonly(info)

console.log(state1);

// 2.传入reactive对象
const state = reactive({
  name: "why",
  age: 18
})
const state2 = readonly(state);

// 3.传入ref对象
const nameRef = ref("why");
const state3 = readonly(nameRef);
```

readonly的应用

- 那么这个readonly有什么用呢？
  - 在我们传递给其他组件数据时，往往希望其他组件使用我们传递的内容，但是不允许它们修改时，就可以使用readonly了；





Reactive判断的API

- **isProxy**
  - 检查对象是否是由 reactive 或 readonly创建的 proxy。
- **isReactive**
  - 检查对象是否是由 reactive创建的响应式代理：
  - 如果该代理是 readonly 建的，但包裹了由 reactive 创建的另一个代理，它也会返回 true；
- **isReadonly**
  - 检查对象是否是由 readonly 创建的只读代理。
- **toRaw**
  - 返回 reactive 或 readonly 代理的原始对象（**不**建议保留对原始对象的持久引用。请谨慎使用）。
- **shallowReactive**
  - 创建一个响应式代理，它跟踪其自身 property 的响应性，但不执行嵌套对象的深层响应式转换 (深层还是原生对象)。
- **shallowReadonly**
  - 创建一个 proxy，使其自身的 property 为只读，但不执行嵌套对象的深度只读转换（深层还是可读、可写的）。

toRefs

- 如果我们使用**ES6的解构语法**，对**reactive返回的对象进行解构获取值**，那么之后无论是**修改结构后的变量**，还是**修改reactive返回的state对象**，**数据都不再是响应式**的：

```
const state = reactive({
  name: "why",
  age: 18
});

const { name, age } = state;
```

- 那么有没有办法**让我们解构出来的属性是响应式**的呢？
  - Vue为我们提供了一个toRefs的函数，可以将reactive返回的对象中的属性都转成ref；
  - 那么我们再次进行结构出来的 name 和 age 本身都是 ref的；

```
// 当我们这样来做的时候，会返回两个ref对象，它们是响应式的
const { name, age } = toRefs(state);
```

- 这种做法相当于已经在state.name和ref.value之间建立了 链接，任何一个修改都会引起另外一个变化；

toRef

- 如果我们只希望转换一个**reactive对象中的属性为ref**, 那么可以**使用toRef的方法**：

```
// 如果我们只希望转换一个reactive对象中的属性为ref，那么可以使用toRef的方法
const name = toRef(state, 'name');
const {age} = state;
const changeName = () => state.name = "coderwhy";
```

ref其他的API

- **unref**
- 如果我们想要**获取一个ref引用中的value**，那么也可以**通过unref方法**：
  - 如果参数是一个 ref，则返回内部值，否则返回参数本身；
  - 这是 `val = isRef(val) ? val.value : val` 的语法糖函数；
- **isRef**
  - 判断值是否是一个ref对象。
- **shallowRef**
  - 创建一个浅层的ref对象；
- **triggerRef**
  - 手动触发和 shallowRef 相关联的副作用：

```
const info = shallowRef({name: "why"});

// 下面的修改不是响应式的
const changeInfo = () => {
  info.value.name = "coderwhy"
  // 手动触发
  triggerRef(info);
};
```

customRef

- 创建一个**自定义的ref**，并**对其依赖项跟踪和更新触发**进行**显示控制**：
  - 它需要一个工厂函数，该函数接受 track 和 trigger 函数作为参数；
  - 并且应该返回一个带有 get 和 set 的对象；

- **这里我们使用一个的案例：**
  - 对双向绑定的属性进行debounce(节流)的操作；

customRef的案例

```
import { customRef } from 'vue';

export function useDebouncedRef(value, delay = 200) {
  let timeout;
  return customRef((track, trigger) => {
    return {
      get() {
        track();
        return value;
      },
      set(newValue) {
        clearTimeout(timeout);
        timeout = setTimeout(() => {
          value = newValue;
          trigger();
        }, delay);
      }
    }
  })
}
```

```
<template>
  <div>
    <input v-model="message">
    <h2>{{message}}</h2>
  </div>
</template>

<script>
  import { useDebouncedRef } from '../hooks/useDebounceRef';

  export default {
    setup() {
      const message = useDebouncedRef("Hello World");
      return {
        message
      }
    }
  }
</script>
```

computed

- 在前面我们讲解过计算属性computed：当我们的某些属性是依赖其他状态时，我们可以使用计算属性来处理
  - 在前面的Options API中，我们是使用computed选项来完成的；
  - 在Composition API中，我们可以在 setup 函数中使用 computed 方法来编写一个计算属性；
- 如何使用computed呢？
  - 方式一：接收一个getter函数，并为 getter 函数返回的值，返回一个不变的 ref 对象；
  - 方式二：接收一个具有 get 和 set 的对象，返回一个可变的（可读写）ref 对象；

```
const fullName = computed(() => {
  return firstName.value + " " + lastName.value;
})
```

```
const fullName = computed({
  get: () => {
    return firstName.value + " " + lastName.value;
  },
  set: newValue => {
    const names = newValue.split(" ");
    firstName.value = names[0];
    lastName.value = names[1];
  }
})
```

侦听数据的变化

- 在前面的Options API中，我们可以通过watch选项来侦听data或者props的数据变化，当数据变化时执行某一些操作。
- 在Composition API中，我们可以使用watchEffect和watch来完成响应式数据的侦听；
  - watchEffect用于自动收集响应式数据的依赖；
  - watch需要手动指定侦听的数据源；

watchEffect

- 当侦听到某些响应式数据变化时，我们希望执行某些操作，这个时候可以使用 watchEffect。
- 我们来看一个案例：
  - 首先，watchEffect传入的函数会被立即执行一次，并且在执行的过程中会收集依赖；
  - 其次，只有收集的依赖发生变化时，watchEffect传入的函数才会再次执行；

```
const name = ref("why");
const age = ref(18);

watchEffect(() => {
  console.log("watchEffect执行~", name.value, age.value);
})
```

watchEffect的停止侦听

- 如果在发生某些情况下，我们希望停止侦听，这个时候我们可以获取watchEffect的返回值函数，调用该函数即可。
- 比如在上面的案例中，我们age达到20的时候就停止侦听：

```
const stopWatch = watchEffect(() => {
  console.log("watchEffect执行~", name.value, age.value);
});

const changeAge = () => {
  age.value++;
  if (age.value > 20) {
    stopWatch();
  }
};
```

watchEffect清除副作用

- 什么是清除副作用呢？
  - 比如在开发中我们需要在侦听函数中执行网络请求，但是在网络请求还没有达到的时候，我们停止了侦听器，或者侦听器侦听函数被再次执行了。
  - 那么上一次的网络请求应该被取消掉，这个时候我们就可以清除上一次的副作用；
- 在我们给watchEffect传入的函数被回调时，其实可以获取到一个参数：onInvalidate
  - 当**副作用即将重新执行** 或者 **侦听器被停止** 时会执行该函数传入的回调函数；
  - 我们可以在传入的回调函数中，执行一些清楚工作；

```
const stopWatch = watchEffect((onInvalidate) => {
  console.log("watchEffect执行~", name.value, age.value);
  const timer = setTimeout(() => {
    console.log("2s后执行的操作");
  }, 2000);
  onInvalidate(() => {
    clearTimeout(timer);
  });
});
```

setup中使用ref

- 在讲解 watchEffect执行时机之前，我们先补充一个知识：在setup中如何使用ref或者元素或者组件？
  - 其实非常简单，我们只需要定义一个ref对象，绑定到元素或者组件的ref属性上即可；

watchEffect的执行时机

- 默认情况下，组件的更新会在副作用函数执行之前：
  - 如果我们希望在副作用函数中获取到元素，是否可行呢？

```
export default {
  setup() {
    const titleRef = ref(null)
    const counter = 0;

    watchEffect(() => {
      console.log(titleRef.value);
    })

    return {
      titleRef,
      counter
    }
  }
}
```

```
null
  <h2></h2>
```

- 我们会发现打印结果打印了两次：
  - 这是因为setup函数在执行时就会立即执行传入的副作用函数，这个时候DOM并没有挂载，所以打印为null；
  - 而当DOM挂载时，会给title的ref对象赋值新的值，副作用函数会再次执行，打印出来对应的元素；

调整watchEffect的执行时机

- 如果我们希望在第一次的时候就打印出来对应的元素呢？
  - 这个时候我们需要改变副作用函数的执行时机；
  - 它的默认值是pre，它会在元素 挂载 或者 更新 之前执行；
  - 所以我们会先打印出来一个空的，当依赖的title发生改变时，就会再次执行一次，打印出元素；
- 我们可以设置副作用函数的执行时机：

```
let h2ElContent = null;

watchEffect(() => {
  h2ElContent = titleRef.value && titleRef.value.textContent;
  console.log(h2ElContent, counter.value);
}, {
  flush: "post"
})
```

- flush 选项还接受 sync，这将强制效果始终同步触发。然而，这是低效的，应该很少需要。

Watch的使用

- watch的API完全等同于组件watch选项的Property：
  - watch需要侦听特定的数据源，并在回调函数中执行副作用；
  - 默认情况下它是惰性的，只有当被侦听的源发生变化时才会执行回调；
- 与watchEffect的比较，watch允许我们：
  - 懒执行副作用（第一次不会直接执行）；
  - 更具体的说明当哪些状态发生变化时，触发侦听器的执行；
  - 访问侦听状态变化前后的值；

侦听单个数据源

- watch侦听函数的数据源有两种类型：
  - 一个getter函数：但是该getter函数必须引用可响应式的对象（比如reactive或者ref）；
  - 直接写入一个可响应式的对象，reactive或者ref（比较常用的是ref）；

```
const state = reactive({
  name: "why",
  age: 18
})

watch(() => state.name, (newValue, oldValue) => {
  console.log(newValue, oldValue);
})

const changeName = () => {
  state.name = "coderwhy"
}
```

```
const name = ref("kobe")

watch(name, (newValue, oldValue) => {
  console.log(newValue, oldValue);
})

const changeName = () => {
  name.value = "james";
}
```

侦听多个数据源

- 侦听器还可以使用数组同时侦听多个源：

```
const name = ref("why");
const age = ref(18)

const changeName = () => {
  name.value = "james";
}

watch([name, age], (newValues, oldValues) => {
  console.log(newValues, oldValues);
})
```

侦听响应式对象

- 如果我们希望侦听一个数组或者对象，那么可以使用一个getter函数，并且对可响应对象进行解构：

```
const names = reactive(["abc", "cba", "nba"]);
watch(() => [...names], (newValue, oldValue) => {
  console.log(newValue, oldValue);
})
const changeName = () => {
  names.push("why");
}
```

watch的选项

- 如果我们希望侦听一个深层的侦听，那么依然需要设置 deep 为true：
  - 也可以传入 immediate 立即执行；

```
const state = reactive({
  name: "why",
  age: 18,
  friend: {
    name: "kobe"
  }
})

watch(() => state, (newValue, oldValue) => {
  console.log(newValue, oldValue);
}, {deep: true, immediate: true})

const changeName = () => {
  state.friend.name = "aaa";
}
```

生命周期钩子

- 我们前面说过 setup 可以用来替代 data 、 methods 、 computed 、watch 等等这些选项，也可以替代 生命周期钩子。
- 那么setup中如何使用生命周期函数呢？
  - 可以使用直接导入的 onX 函数注册生命周期钩子；

```
onMounted(() => {
  console.log("onMounted")
})

onUpdated(() => {
  console.log('onUpdate')
})

onUnmounted(() => {
  console.log('onUnmounted')
})
```

选项式 API	Hook inside `setup`
`beforeCreate`	Not needed*
`created`	Not needed*
`beforeMount`	`onBeforeMount`
`mounted`	`onMounted`
`beforeUpdate`	`onBeforeUpdate`
`updated`	`onUpdated`
`beforeUnmount`	`onBeforeUnmount`
`unmounted`	`onUnmounted`
`activated`	`onActivated`
`deactivated`	`onDeactivated`

TIP

因为 `setup` 是围绕 `beforeCreate` 和 `created` 生命周期钩子运行的，所以不需要显式地定义它们。换句话说，在这些钩子中编写的任何代码都应该直接在 `setup` 函数中编写。

Provide函数

- 事实上我们之前还学习过Provide和Inject，Composition API也可以替代之前的 Provide 和 Inject 的选项。
- 我们可以通过 provide来提供数据：
  - 可以通过 provide 方法来定义每个 Property；
  - provide可以传入两个参数：
    - ✓ name：提供的属性名称；
    - ✓ value：提供的属性值；

```
let counter = 100
let info = {
  name: "why",
  age: 10
}

provide("counter", counter)
provide("info", info)
```

Inject函数

- 在 后代组件 中可以通过 inject 来注入需要的属性和对应的值：
  - 可以通过 inject 来注入需要的内容；
  - inject可以传入两个参数：
    - ✓ 要 inject 的 property 的 name；
    - ✓ 默认值；

```
const counter = inject("counter")
const info = inject("info")
```

数据的响应式

- 为了增加 provide 值和 inject 值之间的响应性，我们可以在 provide 值时使用 ref 和 reactive。

```
let counter = ref(100)
let info = reactive({
  name: "why",
  age: 18
})
provide("counter", counter)
provide("info", info)
```

修改响应式Property

- 如果我们需要修改可响应的数据，那么最好是在数据提供的位置来修改：
  - 我们可以将修改方法进行共享，在后代组件中进行调用；

```
const changeInfo = () => {
  info.name = "coderwhy"
}
provide("changeInfo", changeInfo)
```

useCounter

- 我们先来对之前的counter逻辑进行抽取：

```
import { ref } from 'vue'

export function useCounter() {
  const counter = ref(0);

  const increment = () => counter.value++
  const decrement = () => counter.value--

  return {
    counter,
    increment,
    decrement
  }
}
```

useTitle

- 我们编写一个修改title的Hook：

```
import { ref, watch } from 'vue'

export function useTitle(title = '默认值') {
  const titleRef = ref(title);

  watch(titleRef, (newValue) => {
    document.title = newValue;
  }, {
    immediate: true
  })

  return titleRef;
}
```

useScrollPosition

- 我们来完成一个监听界面滚动位置的Hook：

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

export function useScrollPosition() {
  const scrollX = ref(0)
  const scrollY = ref(0)

  document.addEventListener('scroll', () => {
    scrollX.value = window.scrollX
    scrollY.value = window.scrollY
  })

  return { scrollX, scrollY }
}
```

useMousePosition

- 我们来完成一个监听鼠标位置的Hook：

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

export function useMousePosition() {
  const mouseX = ref(0)
  const mouseY = ref(0)

  window.addEventListener('mousemove', (event) => {
    mouseX.value = event.pageX
    mouseY.value = event.pageY
  })

  return { mouseX, mouseY }
}
```

useLocalStorage

- 我们来完成一个使用 localStorage 存储和获取数据的Hook：

```
import { ref, watch } from "vue"

export function useLocalStorage(key, defaultValue) {
  const data = ref(defaultValue)

  if (defaultValue) {
    window.localStorage.setItem(key, JSON.stringify(defaultValue))
  } else {
    data.value = JSON.parse(window.localStorage.getItem(key))
  }

  watch(data, () => {
    window.localStorage.setItem(key, JSON.stringify(data.value))
  })

  return data;
}
```