## Vue3.0新特性

### 、2) 性能提升:

* 打包大小减少41%
* 初次渲染快55%, 更新渲染快133%
* 内存减少54%
* 使用Proxy代替defineProperty实现数据响应式
* 重写虚拟DOM的实现和Tree-Shaking

### 3) 新增特性

****Composition (组合) API****

setup

* + ref 和 reactive
  + computed 和 watch
  + 新的生命周期函数
  + provide与inject

新组件

* + Fragment - 文档碎片
  + Teleport - 瞬移组件的位置
  + Suspense - 异步加载组件的loading界面

其它API更新

* + 全局API的修改
  + 将原来的全局API转移到应用对象
  + 模板语法变化

问题？

一、使用export default defineComponent 有什么作用

1引入 defineComponent() 以正确推断 setup() 组件的参数类型

2defineComponent 可以正确适配无 props、数组 props 等形式

3defineComponent 可以接受显式的自定义 props 接口或从属性验证对象中自动推断

4在 tsx 中，element-ui 等全局注册的组件依然要用 kebab-case 形式

5 在 tsx 中，v-model 要用 model={{ value, callback }} 写法

6在 tsx 中，scoped slots 要用 scopedSlots={{ foo: (scope) => (<Bar/>) }} 写法

7defineComponent 并不适用于函数式组件，应使用 RenderContext<interface> 解决

# **1. Composition API(常用部分)**

<https://composition-api.vuejs.org/zh/api.html>

### 1) setup

- 新的option, 所有的组合API函数都在此使用, 只在初始化时**执行一次**

- 函数如果返回对象, 对象中的属性或方法, 模板中可以直接使用

### 2) ref

- 作用: 定义一个数据的响应式

- 语法: const xxx = ref(initValue):

- 创建一个包含响应式数据的引用(reference)对象

**- js中操作数据: xxx.value**

**- 模板中操作数据: 不需要.value**

- 一般用来定义一个**基本类型**的响应式数据

<template>

  <h2>{{count}}</h2>

  <hr>

  <button @click="update">更新</button>

</template>

<script>

import {ref} from 'vue'

export default {

  /\* 在Vue3中依然可以使用data和methods配置, 但建议使用其新语法实现 \*/

  /\* 使用vue3的composition API \*/

  setup () {

    // 定义响应式数据 ref对象

    const count = ref(1)

    console.log(count)

    // 更新响应式数据的函数

    function update () {

      // alert('update')

      count.value = count.value + 1

    }

    return {

      count,

      update

    }

  }

}

</script>

### 3) reactive

- 作用: 定义多个数据的响应式

- const proxy = reactive(obj): 接收一个普通对象然后返回该普通对象的响应式代理器对象

- 响应式转换是“深层的”：会影响对象内部所有嵌套的属性

- 内部基于 ES6 的 Proxy 实现，通过代理对象操作源对象内部数据都是响应式的

<template>

  <h2>name: {{state.name}}</h2>

  <h2>age: {{state.age}}</h2>

  <hr>

  <button @click="update">更新</button>

</template>

<script>

import { reactive,} from 'vue'

export default {

  setup () {

    const obj = {

      name: '小明',

      age: 20,

      wife: {

        name: '小甜甜',

        age: 18,

        cars: ['奔驰', '宝马', '奥迪'],

      },

    }

    const state = reactive(obj)

    console.log(state, state.wife)

    const update = () => {

      state.name += '--'

      state.age += 1

    }

    return {

      state,

      update,

    }

  }

}

</script>

### 4) 比较Vue2与Vue3的响应式(重要)

vue2的响应式

- 核心:

- 对象: 通过defineProperty对对象的已有属性值的读取和修改进行劫持(监视/拦截)

- 数组: 通过重写数组更新数组一系列更新元素的方法来实现元素修改的劫持

Object.defineProperty(data, 'count', {

get () {},

set () {}

})

- 问题

- 对象直接新添加的属性或删除已有属性, 界面不会自动更新

- 直接通过下标替换元素或更新length, 界面不会自动更新 arr[1] = {}

Vue3的响应式

- 核心:

- 通过Proxy(代理): 拦截对data任意属性的任意(13种)操作, 包括属性值的读写, 属性的添加, 属性的删除等

- 通过 Reflect(反射): 动态对被代理对象的相应属性进行特定的操作

new Proxy(data, {

    // 拦截读取属性值

    get (target, prop) {

        return Reflect.get(target, prop)

    },

    // 拦截设置属性值或添加新属性

    set (target, prop, value) {

        return Reflect.set(target, prop, value)

    },

    // 拦截删除属性

    deleteProperty (target, prop) {

        return Reflect.deleteProperty(target, prop)

    }

})

proxy.name = 'tom'

### 5) setup细节

- setup执行的时机

- 在beforeCreate之前执行(一次), 此时组件对象还没有创建

- this是undefined, 不能通过this来访问data/computed/methods / props

- 其实所有的composition API相关回调函数中也都不可以

- setup的返回值

- 一般都返回一个对象: 为模板提供数据, 也就是模板中可以直接使用此对象中的所有属性/方法

- 返回对象中的属性会与data函数返回对象的属性合并成为组件对象的属性

- 返回对象中的方法会与methods中的方法合并成功组件对象的方法

- 如果有重名, setup优先

- 注意:

- 一般不要混合使用: methods中可以访问setup提供的属性和方法, 但在setup方法中不能访问data和methods

- setup不能是一个async函数: 因为返回值不再是return的对象, 而是promise, 模板看不到return对象中的属性数据

- setup的参数

- setup(props, context) / setup(props, {attrs, slots, emit})

- props: 包含props配置声明且传入了的所有属性的对象

- attrs: 包含没有在props配置中声明的属性的对象, 相当于 this.$attrs

- slots: 包含所有传入的插槽内容的对象, 相当于 this.$slots

- emit: 用来分发自定义事件的函数, 相当于 this.$emit

在vue3中,$listeners作为$attrs的一部分传递，被包含在内了。

用法：

在B组件（子组件）

    <Item

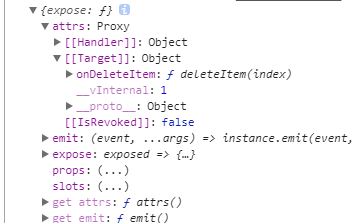
      :todoItem="todo"

      v-bind="$attrs"

    />

在C组件（孙组件）

打印setup中的参数context 可以在attrs中查看到传递过来的找个事件



然后通过context中的emit 去触发即可

    const remove = () => {

      content.emit("deleteItem");

    };

**案例代码另备**

### 6) reactive与ref-细节

- 是Vue3的 composition API中2个最重要的响应式API

- ref用来处理基本类型数据, reactive用来处理对象(递归深度响应式)

- 如果用ref对象/数组, 内部会自动将对象/数组转换为reactive的代理对象

- ref内部: 通过给value属性添加getter/setter来实现对数据的劫持

- reactive内部: 通过使用Proxy来实现对对象内部所有数据的劫持, 并通过Reflect操作对象内部数据

- ref的数据操作: 在js中要.value, 在模板中不需要(内部解析模板时会自动添加.value)

### 7) 计算属性与监视

- computed函数:

- 与computed配置功能一致

- 默认只有getter

- 有getter和setter

- watch函数

- 与watch配置功能一致

- 监视指定的一个或多个响应式数据, 一旦数据变化, 就自动执行监视回调

- 默认初始时不执行回调, 但可以通过配置immediate为true, 来指定初始时立即执行第一次

- 通过配置deep为true, 来指定深度监视

- watchEffect函数

- 不用直接指定要监视的数据, 回调函数中使用的哪些响应式数据就监视哪些响应式数据

- 默认初始时就会执行第一次, 从而可以收集需要监视的数据

- 监视数据发生变化时回调

**computed**

使用 getter 函数，并为从 getter 返回的值返回一个不变的响应式 [ref](https://vue3js.cn/docs/zh/api/refs-api.html" \l "ref) 对象。

const count = ref(1)

const plusOne = computed(() => count.value + 1)

console.log(plusOne.value) // 2

plusOne.value++ // error

或者，它可以使用具有 get 和 set 函数的对象来创建可写的 ref 对象。

    const count = ref(1)

    const plusOne = computed({

    get: () => count.value + 1,

    set: val => {

        count.value = val - 1

    }

    })

    plusOne.value = 1

    console.log(count.value) // 0

**watch**

    const fullName3 = ref("");

    watch(

      user,

      () => {

        fullName3.value = user.firstName + "-" + user.lastName;

      },

      { immediate: true, deep: true }

    );

        // immediate 默认会执行一次watch,deep 深度监视

**侦听多个数据源**

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

     console.log("====");

})

与 [watchEffect](https://vue3js.cn/docs/zh/guide/reactivity-computed-watchers.html" \l "watcheffect) 比较，watch 允许我们：

* + 懒执行副作用；
  + 更具体地说明什么状态应该触发侦听器重新运行；
  + 访问侦听状态变化前后的值。

**watchEffect**

为了根据响应式状态自动应用和重新应用副作用，我们可以使用 watchEffect 函数。它立即执行传入的一个函数，同时响应式追踪其依赖，并在其依赖变更时重新运行该函数

**停止侦听**

当 watchEffect 在组件的 **[setup()](https://v3.cn.vuejs.org/guide/composition-api-setup.html)** 函数或**[生命周期钩子](https://v3.cn.vuejs.org/guide/composition-api-lifecycle-hooks.html)**被调用时，侦听器会被链接到该组件的生命周期，并在组件卸载时自动停止。

在一些情况下，也可以显式调用返回值以停止侦听：

const stop = watchEffect(() => {

/\* ... \*/

});

// later

stop();

**清除副作用**

有时副作用函数会执行一些异步的副作用，这些响应需要在其失效时清除 (即完成之前状态已改变了) 。所以侦听副作用传入的函数可以接收一个 onInvalidate 函数作入参，用来注册清理失效时的回调。当以下情况发生时，这个失效回调会被触发：

* 副作用即将重新执行时
* 侦听器被停止 (如果在 setup() 或生命周期钩子函数中使用了 watchEffect，则在组件卸载时)

const stop = watchEffect(

(onInvalidate) => {

console.log(person.age);

console.log("watchEffect excuted");

let el = document.querySelector("#age") as HTMLElement;

let txt = el && el.innerText;

console.log(txt);

onInvalidate(() => {

console.log("invalidate");

});

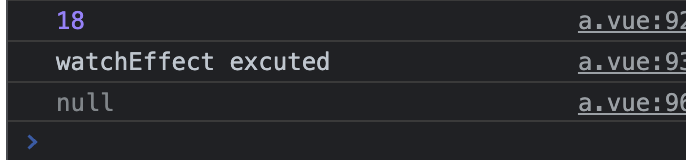
},

{ flush: "post" }

);

**副作用刷新时机**

Vue 的响应性系统会缓存副作用函数，并异步地刷新它们，这样可以避免同一个“tick” 中多个状态改变导致的不必要的重复调用。在核心的具体实现中，组件的 update 函数也是一个被侦听的副作用。当一个用户定义的副作用函数进入队列时，默认情况下，会在所有的组件 update ****前****执行：



默认情况，首次加载获得为null 因为dom还没有加载



拿到到值为之前的。

-----------------------------设置了flush后



数据同步了

### 生命周期

图 看文件夹

beforeCreate -> 使用 setup()

created -> 使用 setup()

beforeMount -> onBeforeMount

mounted -> onMounted

beforeUpdate -> onBeforeUpdate

updated -> onUpdated

beforeDestroy -> onBeforeUnmount

destroyed -> onUnmounted

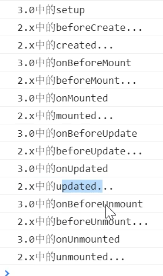
errorCaptured -> onErrorCaptured

新增的钩子函数

组合式 API 还提供了以下调试钩子函数：

onRenderTracked

onRenderTriggered



### 9)自定义hook函数

- 使用Vue3的组合API封装的可复用的功能函数

- 自定义hook的作用类似于vue2中的mixin技术

- 自定义Hook的优势: 很清楚复用功能代码的来源, 更清楚易懂

比如这里自定义一个hook函数

import { ref, onMounted, onUnmounted } from 'vue'

/\*

收集用户鼠标点击的页面坐标

\*/

export default function useMousePosition () {

  // 初始化坐标数据

  const x = ref(-1)

  const y = ref(-1)

  // 用于收集点击事件坐标的函数

  const updatePosition = (e: MouseEvent) => {

    x.value = e.pageX

    y.value = e.pageY

  }

  // 挂载后绑定点击监听

  onMounted(() => {

    document.addEventListener('click', updatePosition)

  })

  return {x, y}

}

使用hook函数

<template>

<div>

  <h2>x: {{x}}, y: {{y}}</h2>

</div>

</template>

<script>

import useMousePosition from './hooks/useMousePosition'

export default {

  setup() {

    const {x, y} = useMousePosition()

    return {

      x,

      y,

    }

  }

}

</script>

### 10) toRefs

把一个响应式对象转换成普通对象，该普通对象的每个 property 都是一个 ref

应用: 当从合成函数返回响应式对象时，toRefs 非常有用，这样消费组件就可以在不丢失响应式的情况下对返回的对象进行分解使用

问题: reactive 对象取出的所有属性值都是非响应式的

解决: 利用 toRefs 可以将一个响应式 reactive 对象的所有原始属性转换为响应式的 ref 属性

<template>

  <h3>foo: {{ foo }}</h3>

  <h3>bar: {{ bar }}</h3>

</template>

<script lang="ts">

import { reactive, toRefs } from "vue";

export default {

  setup() {

    const state = reactive({

      foo: "a",

      bar: "b",

    });

    const stateAsRefs = toRefs(state);

    state.foo += "++";

    state.bar += "++";

    return {

      // ...state,

      ...stateAsRefs,

    };

  },

};

</script>

### 10) toRef

可以用来为源响应式对象上的 property 性创建一个 ref。然后可以将 ref 传递出去，从而保持对其源 property 的响应式连接。

- 为源响应式对象上的某个属性创建一个 ref对象, 二者内部操作的是同一个数据值, 更新时二者是同步的

- 区别ref: 拷贝了一份新的数据值单独操作, 更新时相互不影响

- 应用: 当要将 某个prop 的 ref 传递给复合函数时，toRef 很有用

<template>

  <h2>Child子级组件</h2>

  <h3>age:{{ age }}</h3>

  <h3>length:{{ length }}</h3>

</template>

<script lang="ts">

import { defineComponent, computed, Ref, toRef } from 'vue'

function useGetLength(age: Ref) {

  return computed(() => {

    return age.value.toString().length

  })

}

export default defineComponent({

  name: 'Child',

  props: {

    age: {

      type: Number,

      required: true, // 必须的

    },

  },

  setup(props) {

    const length = useGetLength(toRef(props, 'age'))

    return {

      length,

    }

  },

})

</script>

### 11) ref获取元素

利用ref函数获取组件中的标签元素

功能需求: 让输入框自动获取焦点

<template>

  <div>

    <input type="text" ref="inputRef" />

    <input type="text" ref="inputRef3" id="rr" />

  </div>

</template>

<script lang="ts">

import { defineComponent, onMounted, ref, shallowRef } from "vue";

export default defineComponent({

  name: "App",

  setup() {

    // 默认是空的,页面加载完毕,说明组件已经存在了,获取文本框元素

    const inputRef = ref<HTMLElement | null>(null);

    // console.log(inputRef);

    onMounted(() => {

      let rr = document.getElementById("rr") as HTMLInputElement;

      console.log(rr == inputRef3.value);

      rr.value = "sb";

      //   console.dir(rr);

      console.log(inputRef.value);

      //inputRef.value && inputRef.value.focus(); // 自动获取焦点

    });

    return {

      inputRef,

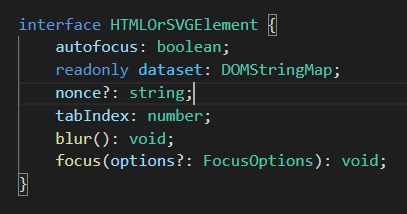
    };

  },

});

</script>

这里的focus 方法 是Ts里面的，因为前面写了类型断言，才能用这个方法，js原生是没有这个方法的。



# **2. Composition API(其它部分)**

### 1) shallowReactive 与 shallowRef

- shallowReactive : 只处理了对象内最外层属性的响应式(也就是浅响应式)

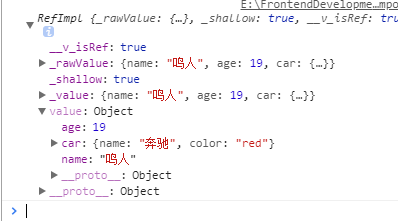
- shallowRef: 只处理了value的响应式, 不进行对象的reactive处理

- 什么时候用浅响应式呢?

- 一般情况下使用ref和reactive即可

- 如果有一个对象数据, 结构比较深, 但变化时只是外层属性变化 ===> shallowReactive

- 如果有一个对象数据, 后面会产生新的对象来替换 ===> shallowRef



### 2) readonly 与 shallowReadonly

- readonly:

- 深度只读数据

- 获取一个对象 (响应式或纯对象) 或 ref 并返回原始代理的只读代理。

- 只读代理是深层的：访问的任何嵌套 property 也是只读的。

- shallowReadonly

- 浅只读数据

- 创建一个代理，使其自身的 property 为只读，但不执行嵌套对象的深度只读转换

- 应用场景:

- 在某些特定情况下, 我们可能不希望对数据进行更新的操作, 那就可以包装生成一个只读代理对象来读取数据, 而不能修改或删除

        const state = reactive({

      name: '佐助',

      age: 20,

      car: {

        name: '奔驰',

        color: 'yellow',

      },

    })

// 只读的数据---深度的只读

     const state2 = readonly(state)

    // 只读的数据---浅只读的

    const state2 = shallowReadonly(state)

### 3) toRaw 与 markRaw

- toRaw

- 返回由 reactive 或 readonly 方法转换成响应式代理的普通对象。

- 这是一个还原方法，可用于临时读取，访问不会被代理/跟踪，写入时也不会触发界面更新。

- markRaw

- 标记一个对象，使其永远不会转换为代理。返回对象本身

- 应用场景:

- 有些值不应被设置为响应式的，例如复杂的第三方类实例或 Vue 组件对象。

- 当渲染具有不可变数据源的大列表时，跳过代理转换可以提高性能。

    const state = reactive<UserInfo>({

      name: '小明',

      age: 20,

    })

    const testToRaw = () => {

      // 把代理对象变成了普通对象了,数据变化,界面不变化

      const user = toRaw(state)

      user.name += '=='

    }

    const testMarkRaw = () => {

      // state.likes = ['吃', '喝']

      // state.likes[0] += '=='

      // console.log(state)

      const likes = ['吃', '喝']

      // markRaw标记的对象数据,从此以后都不能再成为代理对象了

      state.likes = markRaw(likes)

      setInterval(() => {

        if (state.likes) {

          state.likes[0] += '='

        }

      }, 1000)

    }

### 5) 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式控制。它需要一个工厂函数，该函数接收 track 和 trigger 函数作为参数，并应返回一个带有 get 和 set 的对象

customRef的返回值是一个ref对象

使用 v-model 使用自定义 ref 实现 debounce 的示例：

<template>

<input v-model="text" />

</template>

<script>

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)

      }

    }

  })

}

export default {

  setup() {

    return {

      text: useDebouncedRef('hello')

    }

  }

}

</script>

### 6) provide 与 inject

- provide和inject提供依赖注入，功能类似 2.x 的provide/inject

- 实现跨层级组件(祖孙)间通信

<template>

  <h1>父组件</h1>

  <p>当前颜色: {{color}}</p>

  <button @click="color='red'">红</button>

  <button @click="color='yellow'">黄</button>

  <button @click="color='blue'">蓝</button>

  <hr>

  <Son />

</template>

<script lang="ts">

import { provide, ref } from 'vue'

import Son from './Son.vue'

export default {

  name: 'ProvideInject',

  components: {

    Son

  },

  setup() {

    const color = ref('red')

    provide('color', color)

    return {

      color

    }

  }

}

</script>

<template>

  <h3 :style="{color}">孙子组件: {{color}}</h3>

</template>

<script lang="ts">

import { inject } from 'vue'

export default {

  setup() {

    const color = inject('color')

    return {

      color

    }

  }

}

</script>

### 7) 响应式数据的判断

- isRef: 检查一个值是否为一个 ref 对象

- isReactive: 检查一个对象是否是由 reactive 创建的响应式代理

- isReadonly: 检查一个对象是否是由 readonly 创建的只读代理

- isProxy: 检查一个对象是否是由 reactive 或者 readonly 方法创建的代理

# 3.新组件

### 1) Fragment(片断)

- 在Vue2中: 组件必须有一个根标签

- 在Vue3中: 组件可以没有根标签, 内部会将多个标签包含在一个Fragment虚拟元素中

- 好处: 减少标签层级, 减小内存占用

<template>

    <h2>aaaa</h2>

    <h2>aaaa</h2>

</template>

### 2) Teleport(瞬移)

Teleport 提供了一种干净的方法, 让组件的html在父组件界面外的特定标签(很可能是body)下插入显示

<template>

  <button @click="modalOpen = true">打开一个对话框</button>

  <!--对话框代码-->

  <Teleport to="body">

    <div v-if="modalOpen" class="modal">

      <div>

        这是对话框

        <button @click="modalOpen = false">关闭对话框</button>

      </div>

    </div>

  </Teleport>

</template>



### 3) Suspense(不确定的)

它们允许我们的应用程序在等待异步组件时渲染一些后备内容，可以让我们创建一个平滑的用户体验

<template>

  <h2>App父级组件:Suspense组件的使用</h2>

  <Suspense>

    <template #default>

      <!--异步组件-->

      <!-- <AsyncComponent /> -->

      <AsyncAddress />

    </template>

    <template v-slot:fallback>

      <!--loading的内容-->

      <h2>Loading.....</h2>

    </template>

  </Suspense>

</template>

<script lang="ts">

import { defineComponent, defineAsyncComponent } from 'vue'

// 引入组件:静态引入和动态引入

// Vue2中的动态引入组件的写法:(在Vue3中这种写法不行)

// const AsyncComponent = () => import('./AsyncComponent.vue')

// Vue3中的动态引入组件的写法

// const AsyncComponent = defineAsyncComponent(

//   () => import('./AsyncComponent.vue')

// )

// 静态引入组件

// import AsyncComponent from './AsyncComponent.vue'

import AsyncAddress from './AsyncAddress.vue'

export default defineComponent({

  name: 'App',

  components: {

    // AsyncComponent,

    AsyncAddress,

  },

})

</script>

<template>

  <h2>AsyncAddress组件</h2>

  <h3>{{ data }}</h3>

</template>

<script lang="ts">

import { defineComponent } from 'vue'

// 引入axios

import axios from 'axios'

export default defineComponent({

  name: 'AsyncAddress',

  async setup() {

    const result = await axios.get('/data/address.json')

    return {

      data: result.data,

    }

  },

})

</script>

# 深入源码

### proxy为什么配合Reflect使用

使用Reflect.get而不是target[key]的原因是receiver参数可以把this指向getter调用时，而非Proxy构造时的对象。

一句话概括就是：触发代理对象劫持的时候，保证正确this上下文指向

\* Proxy中接受的Receive形参表示的是代理对象本身或者继承于代理对象的对象

\*Reflect中的Receive 表示的是修改执行原始操作时的this指向