# Vue3.0新特性

https://24kcs.github.io/vue3\_study/chapter5/02\_%E5%85%B6%E4%BB%96%E6%96%B0API.html#%E5%85%A8%E6%96%B0%E7%9A%84%E5%85%A8%E5%B1%80api

# 一、基本特性Breaking

1.$attrs 包含class style

2.Vue.prototype = app.config.globalProperties

3.new Vue({}) =>

import { createApp } from 'vue' ; const app = createApp(App) ; app.mount('#app')

4.自定义指令生命周期改变

app.directive('highlight', {

created() { },

// 通过binding获取组件实例

beforeMount(el, binding) {

console.log(binding)

el.style.color = binding.arg

el.style.fontSize = binding.value + 'px'

},

beforeUpdate() { },

beforeUnmount() { }

})

1. 取消了$children，可以用$ref替换
2. 异步组件使用内部api

import { defineAsyncComponent } from "vue";

// 异步组件Vue3.0新写法。

// const asyncModalWithOptions = defineAsyncComponent({

//   loader: () => import('./Modal.vue'),

//   delay: 200,

//   timeout: 3000,

//   errorComponent: ErrorComponent,

//   loadingComponent: LoadingComponent

 components: {

    // 异步组件Vue3.0新语法

    HelloWorld: defineAsyncComponent(() =>

      import("./components/HelloWorld.vue")

    ),

  },

// })

1. 新增 v-is指令，用于展示WEB component API

 <table>

    <tr v-is="'blog-post-row'"></tr>

  </table>

  <button v-is="'blog-post-row'"></button>

1. 取消了过滤器，全局替换可以用app.config.globalProperties.$filters = (...args)=>{},内部可以用computed属性替换
2. 取消了$on $off $once方法，内部可通过用一个tiny-emitter库来实现抛出的功能

import emitter from 'tiny-emitter/instance'export default {

$on: (...args) => emitter.on(...args),

$once: (...args) => emitter.once(...args),

$off: (...args) => emitter.off(...args),

$emit: (...args) => emitter.emit(...args),}

1. 组件可以有根节点，不需要template下包裹一个父div，组件继承的相关介绍。当只有一个根节点时，非props emit属性将继承作用于根节点,比如给子组件添加自定义属性或者class，多跟节点则需要自定义绑定主节点，添加到组件上v-bind=”$attrs”
2. 函数式组件，js形式

import { h } from 'vue'

const DynamicHeading = (props, context) => {

    console.log(props)

    console.log(context)

    return h(`h${props.level}`, context.attrs, context.slots)

}

DynamicHeading.props = ['level']

export default DynamicHeading

单文件组件形式

与 2.x版本相比区别在于，取消了template的functional <template>

<component

v-bind:is="`h${$props.level}`"

v-bind="$attrs"

/></template>

<script>export default {

props: ['level']}</script>

# 二 Global API breaking

1. 取消了new Vue（）的写法，引入全局API createApp

import { createApp } from 'vue'

const app = createApp({})

写法转换

Vue=>app

Vue.prototype = app.config.globalProperties

|  |  |
| --- | --- |
| Vue.config | app.config |
| Vue.config.productionTip | removed ([see below](https://v3.vuejs.org/guide/migration/global-api.html" \l "config-productiontip-removed)) |
| Vue.config.ignoredElements | app.config.isCustomElement ([see below](https://v3.vuejs.org/guide/migration/global-api.html" \l "config-ignoredelements-is-now-config-iscustomelement)) |
| Vue.component | app.component |
| Vue.directive | app.directive |
| Vue.mixin | app.mixin |
| Vue.use | app.use ([see below](https://v3.vuejs.org/guide/migration/global-api.html" \l "a-note-for-plugin-authors)) |
| Vue.prototype | app.config.globalProperties ([see below](https://v3.vuejs.org/guide/migration/global-api.html" \l "vue-prototype-replaced-by-config-globalproperties)) |

provide的新注入。

// in the entry

app.provide('guide', 'Vue 3 Guide')

// in a child componentexport default {

inject: {

book: {

from: 'guide' // from 新参数

}

},

template: `<div>{{ book }}</div>`}

1. 插件的使用

import { nextTick } from 'vue'

const plugin = {

install: app => {

nextTick(() => {

// ...

})

}}

如果您使用像 webpack 这样的模块包，这可能会导致 Vue 的源代码被捆绑到插件中，而且这通常不是您所期望的。防止这种情况发生的常见做法是配置模块捆绑器以将 Vue 从最终捆绑包中排除。在 webpack 的情况下，你可以使用[externals （打开新窗口）](https://webpack.js.org/configuration/externals/" \t "https://v3.vuejs.org/guide/migration/_blank)配置选项：

// webpack.config.js

module.exports = {

/\*...\*/

externals: {

vue: 'Vue'

}}

这将告诉 webpack 将 Vue 模块视为外部库而不是捆绑它。

1. Key的新属性 ， vue3.x 中 key不推荐使用到v-if v-show等标签中，会自动生成，3.x中key应该放在template上而不是子节点上，当v-for与v-if共同使用，key绑定到template上。

注：vue3.x中，v-if优先级大于v-for ， vue2.x中v-for优先级大于v-if

<!-- Vue 2.x -->

<template v-for="item in list">

<div v-if="item.isVisible" :key="item.id">...</div>

<span v-else :key="item.id">...</span></template>

<!-- Vue 3.x -->

<template v-for="item in list" :key="item.id">

<div v-if="item.isVisible">...</div>

<span v-else>...</span></template>

1. 键码修饰符 vue3.x v-on不支持数字，也不支持config.keyCodes形式

<input v-on:keyup.delete="confirmDelete" />

1. 挂载形式不同，vue2.x替换的直接是id=app的根div。Vue3.x替换的是id=app的根div的innerhtml。相当于一个替换掉，一个是插入。

6.propsData vue3.x中使用的是 createApp的第二个参数

import './directives'

const app = createApp(App, {

    name: "wsnd"

})

Vue 2.x中写法，传递props给根组件 （app）

const Comp = Vue.extend({

props: ['username'],

template: '<div>{{ username }}</div>'})

new Comp({

propsData: {

username: 'Evan'

}})

# 核心 breaking

1. 组合API，一个多复用组件的各功能模块代码，组合到setup函数中，此函数接收两个参数:props,context，props本身是属于reactive的，所以不能用es6解构，否则会破坏他的reactive状态，可以通过使用toRefs来实现解构。如果prop中的属性是可选的，只需要让属性中的一项需要为引用的响应式，则需要使用toRef(prps,’args’)来实现解构。响应式函数ref,reactive 来使定义变量是响应式的。setup执行于组件创建实例之前 beforeCreate-creacted之间 ,this 不能访问到所有的data methods等东西，只能获取到props ，setup中可以使用生命周期函数，以onMounted为例，on开头，也可以使用独立的computed，watch等外部api 。理论上组合api不能代替外部api 挤 props watch data methods等。所有的api的引入都需要通过按需引入

import { ref, reactive, watch, computed, onMounted } from "vue";

export default {

  name: "App",

  provide: {},

  setup(props, context) {

    console.log(props,'props');

    console.log(context,'context');

    let count = ref(0);

    let obj = reactive({

      title: "reactive",

    });

    console.log(count.value);

    return {

      count,

      obj,

    };

  },

1. 组合api中使用provide inject ，提供provide的顶层组件中

import { provide } from 'vue'import MyMarker from './MyMarker.vue'

export default {

components: {

MyMarker

},

setup() {

provide('location', 'North Pole')

provide('geolocation', {

longitude: 90,

latitude: 135

})

}}

接受inject的子组件中

<script>import { inject } from 'vue'

export default {

setup() {

const userLocation = inject('location', 'The Universe')

const userGeolocation = inject('geolocation')

return {

userLocation,

userGeolocation

}

}}</script>

理论上，修改proveide所提供的值，最好是在提供者本身中修改。如果需要在子组件中修改，则可以通过顶层组件中proveide一个fn到子组件中修改掉值。一般来说provide的值都不是响应式，可通过reactive或者ref来改变为响应式。可以在顶层中引入readOnly来防止子组件中修改掉其值

// shallowRef 不会对其转换为响应式，但会追踪它的value变化，即使变更后的值也不会转变为响应式

<script>import { provide, reactive, readonly, ref } from 'vue'import MyMarker from './MyMarker.vue'

export default {

components: {

MyMarker

},

setup() {

const location = ref('North Pole')

const geolocation = reactive({

longitude: 90,

latitude: 135

})

const updateLocation = () => {

location.value = 'South Pole'

}

provide('location', readonly(location))

provide('geolocation', readonly(geolocation))

provide('updateLocation', updateLocation)

}}</script>

1. V-bind 和 单个属性的优先级合并问题。Vue2.x 单个属性将永远覆盖v-bind的obj属性。 vue3.x根据绑定顺序进行优先合并。

2.x:

<!-- template -->

<div id="red" v-bind="{ id: 'blue' }"></div>

<!-- result -->

<div id="red"></div>

3.x:

<!-- template -->

<div id="red" v-bind="{ id: 'blue' }"></div>

<!-- result -->

<div id="blue"></div>

<!-- template -->

<div v-bind="{ id: 'blue' }" id="red"></div>

<!-- result -->

<div id="red"></div>

1. VNode生命周期 vue2.x中使用如@hook-updated，hook开头的周期事件监听组件的生命周期中的关键阶段，vue3.x中开用@vnode-updated，以vnode开头。

<template>

<child-component @hook:updated="onUpdated">

</template>

<template>

<child-component @vnode-updated="onUpdated">

</template>

驼峰也是可以允许的。

<template>

<child-component @vnodeUpdated="onUpdated">

</template>

#### toRef toRefs ref reactive区别

##### Reactive接受一个对象，改为响应式。 ref更灵活，可以直接赋值，但本质上还是以reactive的形式。

 ref和toRef区别：

 ref->复制，修改响应式数据不会影响以前的数据

toRef->引用，修改响应式数据会影响以前的数据

ref->数据发生改变，界面就会自动更新

toRef->数据发生改变，界面也不会自动更新

##### toRef(obj,’args’) 哪个对象，哪个属性

##### toRefs(obj)哪个对象的全部属性

 toRef应用场景：

如果想让响应式数据和以前的数据关联起来，并且更新响应式数据之后还不想更新UI，那么久可以使用toRef

# vuex vue-router的使用

1. creactStore CreactRouter createWebHashHistory 等新api的引入。
2. import { createRouter, createWebHashHistory } from 'vue-router';
3. import Home from "../views/Home.vue";
4. const routes = {
5. {
6. path: "/",
7. name: "Home",
8. component: Home,
9. },
10. {
11. path: "/about",
12. name: "About",
13. // route level code-splitting
14. // this generates a separate chunk (about.[hash].js) for this route
15. // which is lazy-loaded when the route is visited.
16. component: () =>
17. import(/\* webpackChunkName: "about" \*/ "../views/About.vue"),
18. }
19. }
20. const router = createRouter({
21. history: createWebHashHistory(),
22. routes
23. })
24. export default router

import Vuex from 'vuex'

export default Vuex.createStore({

state: {

},

mutations: {

},

actions: {

},

modules: {

}

});

Vue组件实例中，取消了this ，可以通过 getCurrentInstance来获取到来替换this，可以通过ctx来使用ctx.$router $store等方式。

批注： 1.当把ref值赋值给reactive的值时，会自动解包。

const count = ref(1)

const obj = reactive({})

obj.count = count // 1

# 核心总结

1.setup(props,context) setup就是beforeCreate周期，获取不到this ， 必须return一个对象， 对象中的属性合并到data, 方法return到methods，同名以setup中的为主

###### 2. ref 和 reactive区别 ， ref是用来基本属性的定义，内部是给value属性添加getter setter 来劫持，reactive是用的proxy代理来劫持，可以定义一些复杂属性，如object， 如果ref定义的也是object等，则会在内部自动转为reactive来劫持

# **三 、 Vue3.0 + TypeScript**

### 1 . Interface 接口，做类参函数等的限制

// 接口，做限制类

interface A {

    name: string,

    age: number

}

function showFullName(person: A): string {

    return person.name + person.age

}

const person = {

    name: '20',

age: 20

// 必须符合类A的限制

}

showFullName(person)

### 定义数组

Let arr : number[] = []

let arr :number [string,number] = [“1”, 0]

Let arr: Array<number>=[]

# 枚举

// 值从0开始， A = 1

Emum A {

A,

B,

C

}

字符串枚举，必须为每个key赋值

Emum A {

A=”A“,

B = “B”,

C = “C”

}

1. 联合类型 let a:string|number 可以为string，可以为number
2. 类型断言 类型断言，告诉编译器，我这个是肯定是什么， (<string>arg) 告诉编译器我确定我这个arg参数是string的，可以直接执行后面的逻辑

方式一: <类型>值

方式二: 值 as 类型 tsx中只能用这种方式

/\* 需求: 定义一个函数得到一个字符串或者数值数据的长度

\*/function getLength(x: number | string) {

if ((<string>x).length) {

return (x as string).length

} else {

return x.toString().length

}}console.log(getLength('abcd'), getLength(1234))

1. 类型推断 定义时不赋值，推断any 定义赋值了，就根据值判断

### 8 核心 interface 接口（就是一种规范，约束）

Interface A {} 所有限制类型都必须满足A接口的限 , 接口中的所有的属性都不能有实际值

Type也可以声明一个对象接口限制，区别在于

1. 声明多个interface的时候，最后会合并。而Type不会合并

2. 接口可以在定义类的时候去限制类的解构

// 可以限制普通对象

interface IPerson {

readonly id: number // 只读

name: string

age: number

sex?: string //可选可不选

}

const person2: IPerson = {

id: 2,

name: 'tom',

age: 20,

// sex: '男' // 可以没有

// xxx: 12 // error 没有在接口中定义, 不能有

}

person2.id = 2 // error ， 因为只读，所以不能修改

// 也可以用来约束函数

interface func {

    (name: string, age?: number): string

}

const getname: func = function (name: string): string {

    return name

}

console.log(getname("paul"))

// 限制类class， 类类型

关键字 implements 实现

一个类可以实现多个interface

一个interface也可以继承多个interface

接口接口之间叫继承

类可以实现接口来约束类 ES7 implements

interface A {

    name?: string,

    age: number

}

interface B {

    gender: "boy"

}

interface AA extends A, B {

}

// 实现接口

class demo implements A, B {

    name: string = "1"

    age: number = 10

    gender:string = "123"

}

1. 类 Class super关键字

super以对象的形式调用，代表父类

Super以函数形式调用，代表执行父类的构造函数， 需要传入父类的构造函数的传入的参数，super（）代表的执行父类的构造函数

class A {

    name: string

    age: number

    constructor(name: string, age: number) {

        // this  => this指向当前实例

        this.name = name

        this.age = age

    }

    alert() {

        console.log("我是父类A")

        console.log(this.name + "," + this.age)

    }

}

class B extends A {

    gender: string

    //

    constructor(name: string, age: number, gender: string) {

        super(name, age)

        this.gender = gender

    }

    // constructor()

}

// 抽象类不能被创建对象，以new的形式生成实例。

// 只能被继承 ， 可以添加抽象方法

Abstract Class C {

// 抽象方法以abstrac开头，只能定义在抽象类中，并且继承的子类中必须对此方法重写，否则不能调用

Abstract sayHello():void{

}

}

abstract class A {

    name: string

    age: number

    constructor(name: string, age: number) {

        // this  => this指向当前实例

        this.name = name

        this.age = age

    }

    abstract alert(): void // 抽象类方法没有方法体， 以abstract开头

}

继承的A的子类中抽象方法alert 必须重写，否则报错

TS 独有寄存器， set get 方法

class A {

    public name: string

    private age: number // 私有属性

    protected gender: string // 只能在当前类和子类中使用，！但子类的实例访问不到！

    constructor(name: string, age: number, gender: string) {

        this.name = name

        this.age = age

        this.gender = gender

    }

    public sayHello(a: string) {

        return '123'

    }

    // private 属性外部是无法获取的，但可以通过父类的方法暴露出去，

    // 修改赋值的权限控制在父类中，可以做一些validate

    getAge(): number {

        return this.age

    }

    setAge(val: number): void {

        this.age = val

    }

    // 寄存器

    // ts中独有的set，get方法用来改变父类方法的属性

    get myage() {

        return this.age

    }

    set myage(val: any) {

        this.age = val

    }

}

class B extends A {

}

const b = new B("paul", 20, "girl")

b.myage = 30

console.log(b.name)

使用Public修饰符简写的语法糖

// 使用public简写的语法糖

class C {

    constructor(public name: string, public age: number) {

    }

}

 // 这两种写法效果一致

class D {

    name: string

    age: number

    constructor(name: string, age: number) {

        this.name = name

        this.age = age

    }

}

##### 泛型

定义函数或类不确定类型，执行调用时才确定类型的一种设计模式