# vue3和vue2比较

1）响应式的区别：

vue2 defineProperty

vue3 proxy

2）diff算法

vue2每次diff更新都是全量对比

vue3只对动态的元素（带有标记的）进行比对  
3）vue3可以在template下写多个元素

4）支持tsx和jsx的写法

5）新增了Suspense teleport 和多v-model的用法

6）支持tree shaking

如import {message } from “”

没有用到的其他的属性是不会打包进来的。

7）setup

8）createApp(App).mount()语法

//优点：可以生成多个实例，多人同时开发。

import { createApp } from 'vue'

import App from "./App.vue"

import Root from "./Root.vue"

createApp(Root).mount("#root");

createApp(App).mount("#app");

# 2、vite的特点和优势

1）冷服务，开发环境采用script type=module这种方式引入文件，而webpack则是将文件打包之后再引入。当然在生产环境还是会编译打包的。

# 3、nvm简单说明

1）nvm list：查看本地安装的node的所有版本

2）nvm list available 查看所有可用的node版本

3）nvm install xxx（对应的版本号）安装对应的版本

4）nvm use xxx（对应的版本号）使用对应的版本

原理就是将所有的版本放到某个目录下，切换环境变量来使用不同的版本。

# 4、nrm介绍

1）nrm是一个npm源管理器，允许你快速的在npm源间进行切换。

npm install -g nrm

2）nrm ls 查看所有的可用的源以及对应的仓库地址

3）nrm use xxx（源名称）用于切换源

4）nrm add 名称 仓库地址 添加源

# 5、npm run dev执行过程

执行的是vite命令，利用软连接关联到bin目录下的vite进行执行

# 6、vscode的插件

1）vetur（vue2）与volar（vue3）良好的vue代码提示，使用volar时需要把vetur禁用

# 7、vue3新特性

**#### 异步导入组件**

```js

//======start2=====

//2、异步导入组件

//在打包后会生成一个单独的js文件，而不是直接打包进app.js中。

//（1）局部异步组件

import {defineAsyncComponent} from "vue";

const AppAsync = defineAsyncComponent(() => {

    import ('./components/AppAsync.vue')

})

export default{

    components: {

        AppAsync

    }

}

//（2）全局异步组件

//在main.js中

import {defineAsyncComponent} from "vue";

const AppAsync = defineAsyncComponent(() => {

    import ('./components/AppAsync.vue')

})

createApp(App).component("app-async", AppAsync)//这样在任何组件内就可以直接使用AppAsync了。

//======end========

```

**#### 自定义指令的生命周期**

```js

//3、自定义指令的生命周期

//main.js中

import App from "./App.vue"

const app = createApp(App)

app.mount("#app")

app.directive("focus", {

    beforeMount(el, binding, vnode, preVnode) {

        console.log(binding.instance)

    },

    mounted(el) {

        el.focus()

    },

    unmounted() {

        alert("我被卸载了")

    }

})

// App.vue中

<input v-focus/>

//当v-focus删除的后，触发unmounted函数

//如果在App.vue（也可以是其他的模块中）有如下写法,则input元素会被传送到指定的元素上面。

<teleport to="body">

<input/>

</teleport>

```

**#### data的修改**

```js

//4、data的修改

//data不再接受纯javaScript object，而必须使用function 声明返回

//这样做的好处是如果使用对象的形式，则数据与后文使用的data有共享行为，假如改变某个属性，则会影响后文中对应属性的值

//采用函数的形式的话，则生成一个新的对象，保证data的数据只存在与当前模块中。

export default{

    data: {

        state: ""

    }

}

export default{

    data(){

        return {

        }

    }

}

```

**#### 组件内支持多个节点**

```js

//5、多个根节点的支持

//（不再只是必须只有一个根节点了，但是需要设置默认的根节点）

//有根节点的话props的属性会被挂载到根节点上

<template>

   <div></div>

</template>

//没有根节点的话，需要设置默认根节点(v-bind="$attrs")，这样的话，props就会挂载到指定的默认根节点上。

<template>

 <p v-bind="$attrs"></p>

 <img/>

</template>

```

**#### 函数式组件**

```js

//6、函数式组件

//在vue2中函数式组件只用于无状态值得时候的结构渲染（只有props属性）

//这样渲染的性能会高一些，实例化不会频繁的发生

export default{

    props: {

        msg: "hello world"

    },

    functional: true

    //没有data

}

//vue3的函数式组件的写法

import {h} from "vue"

export default function Functional(props, context) {

  return h('div', {

      context.attrs,

      props.msg

  })

}

Functional.props = ["msg"] //定义此语句的话，context中就不会再有props中的msg属性

```

**#### 其他的一些更新说明**

```js

//7、v-if 和 v-for可以写在同一个标签上，且v-if比v-for的优先级更高。

//8、同一个标签上的同一个属性（props），后者比前者的优先级更高。

```

**#### 颠覆意义的变化（重点）**

**##### setup**

```js

//9、setup函数,watch,reactive,watchEffect

//组件内部

import {reactive, watch, watchEffect} from "vue"

export default{

    setup(props, context) {

    const data = reactive({value: 0, count: 0})

    const increament = () => {

        data.value ++;

    }

    watch(() => { //watch第一个参数如果为函数的话，会以函数的返回值的变化作为副作用执行的条件

    //副作用的参数就是该返回值

        return data.value > 2

    }, (isExceeded) => {

        console.log(isExceeded)

        if(isExceeded) {

            console.log("超过了2")

        }else {

            //这一步不会执行

            console.log(isExceeded)

        }

    })

    watch(data, (value)=> { //第一个参数如果为响应式的对象的话，则以对象的属性变化作为副作用执行的条件

    //副作用的参数就是该响应式对象

        console.log('changed', value)

    }, {immediate: true}) //第三个参数是配置项)

    watchEffect(() => { //只要响应式对象data的属性发生变化，就会执行,注意只有访问data的属性的时候watchEffect才会执行，只访问

    //data是不执行的,watchEffect进入到界面的时候就会执行一次，watch则需要配置选项immediate:true.

      console.log("", data.value)

    })

    return {

        data,

        increament

    }

  }

}

//watchEffect清除副作用oninvalidate，watchEffect会返回一个stop函数

const stop = watchEffect((oninvalidate) => {

console.log(message.name) //这段代码会在oninvalidate之后进行

oninvalidate(() => {console.log(‘before’)})

}, {flush: “post”})//最后一个参数可以配置watch函数的调用时机

const clickToStop = () => stop()执行完这一步以后，就不会再监听message.data了。

```

**##### ref**

```js

//10.ref:用于包装基本类型的值为响应式对象

//<templete>

//<p>{{dataRef}}</p> //界面展示10,不要写dataRef.value

//<button @click="increament">点击</button>

//</template>

import {ref} from "vue";

export default{

    setup() {

        const dataRef = ref(10)

        const increament = () => {

            dataRef.value++ //这里必须写value

        }

    }

    return {

        dataRef,

        increament

    }

}

//customeRef

fucntion MyRef<T>(value:T) {

let timer:any

return customeRef((track, trigger) => {

return {

get() {

track()

}

set(newVal) {

clearTimeout(timer)

timer = setTimeout(() =>{

value = newVal;

trigger()

timer = null;

}, 500)

}

}

})

}

```

shallowRef:浅层响应式如obj1 = shallowRef({name: “zhang”}),修改值的时候，如果

需要值在dom中响应，需要 obj1.value = {name: “wang”},而ref则是obj.value=”wang”;shallowRef 和ref不能同时使用，原因，ref值修改会影响shallowRef,使obj1.value.name = “wang”成为响应式。

triggerRef(obj1)会强制更新收集的依赖：

**##### readonly/reactive/shallowReactive/shellowReadonly**

```js

import {readonly, reactive,computed, shallowReactive, shellowReadonly} from "vue"

//11、readonly用于创建只读对象，也就是说对象中的属性是不可以修改的

const dataReadOnly = readonly({value: "This is value"})

//12、shallowReactive对对象的浅层属性进行响应，深层不会,应用场景：用于在深层的对象只适用于计算，不会应用于页面的响应更新的时候。shalloReactive和shallowRef一样不能和reactive同时使用。

//13、reactive和ref区别：

reactive只适用于对象。

reactive中的值修改直接obj.属性名就行

reactive不能直接赋值。例如let list = reactive<string[]>([]),调用接口返回的值，直接使用list = res;是不行的，页面上没有响应。原因list是一个proxy代理对象，直接赋值的化，会覆盖调代理。解决方法，list.push(...res);

//shellowReadonly浅层的属性不可以进行更改。深层的仍然可以修改。

const shellowReactive = shellowReactive({value: "value", behavior: {run: "", walk: ""}})

//13、toRaw,markRow，toRef,toRefs

//toRaw返回代理对象的原对象

//markRow将普通对象转换为不可代理的对象

//toRef(reactiveObj, key)只能修改响应式对象的值，非响应式对象视图毫无变化。toRef(obj, key),使用场景，当视图上只需要响应式对象的一个属性的时候。

//toRefs(reactiveObj)则是将响应式对象的值都变为响应式的。用于解构赋值。比如obj = reactive({name:”zhang”,age:10}) const {name, age} = toRefs(obj);那么修改在程序中按照以下语法name.value = “hello” 视图上会更新。如果不采用toRefs()的话，这种语法不会导致视图更新。

//14、computed生成衍生值,两种书写方式

const total = computed(() => dataRef.value \*\*2)

const total = computed({get() {return firstName.value + lastName.value},set() {firstName.value + lastName.value}})

```

//15、

**##### provide 和 inject跨组件传递数据的方案**

```js

//父组件中

import InjectAndProvide from "./components/InjectAndProvide.vue"

export default {

  name: 'App',

  provide() {

    return {

      state: "状态",

      onChange() {

          console.log(this.state)

      }

    }

  },

  components: {

    HelloWorld,

    InjectAndProvide

  }

}

//InjectAndProvide组件中

// <button @click="onChange">触发onChange事件</button>

import {inject} from "vue"

export default{

    setup(props, context) {

      const state = inject("state");

      const onChange = inject("onChange");

      return {

          state,

          onChange

      }

    }

}

```

**//16.defineProps 和 defineEmits直接在<script setup></script>中使用，无需导入**

**/父组件中**

 <Hooks name="zhang" change="onChanges">

/子组件中

<template>

    <div>

        <p>{{id}}</p>

        <p>{{props.name}}</p>

        <button @click="emitChange">修改属性</button>

    </div>

</template>

<script setup>

//setup的这种写法

import {ref} from "vue"

const props = defineProps({

    name: String

})

const emits = defineEmits({

    onChanges: Function

})

const id = ref(0);

</script>

//17.hooks

//定义HOOKS文件

const fetch = () => {

return new Promise((resolve, reject) => {

setTimeout(()=>{

resolve(Array(10).fill(0).map((, index) => ({

name: ‘user-’ + Math.random(), id

})))

}, 1000)

})

}

//export default function useUserInfo(id) {

const info = ref({});

const loading= ref(false)

const getUserInfo = () => {

loading.value = true;

fetch(id.value).then((user) => {

info.value = user;

loading.value = false;

})

}

onMounted(getUserInfo)

watch(() => id.value, getUserInfo)

return {loading, info}

}

//使用HOOKS组件

<script setup>

import useUserInfo from “./path”

const id = ref(0)

const {info, loading} = useuserInfo(id)

const changePage =()=> {

id.value++

}

</script>

//======end========

//18、补充知识

const initData = {

    value: 1

}

const proxy = new Proxy(initData,{

    get(target, key) {

        console.log(target[key], "-----")

        return target[key]

    },

    set(target, key, val){

        console.log(`修改了属性${key}`)

        Reflect.set(target, key, val)

    }

})

// console.log(initData.value) //这里不会触发get函数

// console.log(proxy.value) //这里会触发get函数

// initData.value = 10; //不会触发set函数

// console.log(initData.value) // 10

// console.log(proxy.value) //10 触发get函数。

proxy.value = 20; //触发set函数

// proxy.aa = 30; //触发set函数

initData.aa = 40; //不会触发set函数

console.log(proxy.aa) //触发get函数

//Object.defineProperty会直接在对象上定义一个新的属性。

//只有对对象原型上有的属性操作才会触发get和set函数，如果对象原型不存在某个属性，则不会触发get和set函数

const initData = {

    value: 1

}

const data = {

}

Object.keys(initData).forEach((key)=> {

    Object.defineProperty(data, key, {

        get() {

            console.log("访问了", key)

            return initData[key];

        },

        set(value) {

            console.log("ppppp")

            initData[key] = value;

        }

    })

})

//以下代码会触发get函数

console.log(data.value)

//以下代码会触发set函数

data.value = 10;

//以下代码不会触发set函数

data.aa = 20;

//以下代码不会触发get函数

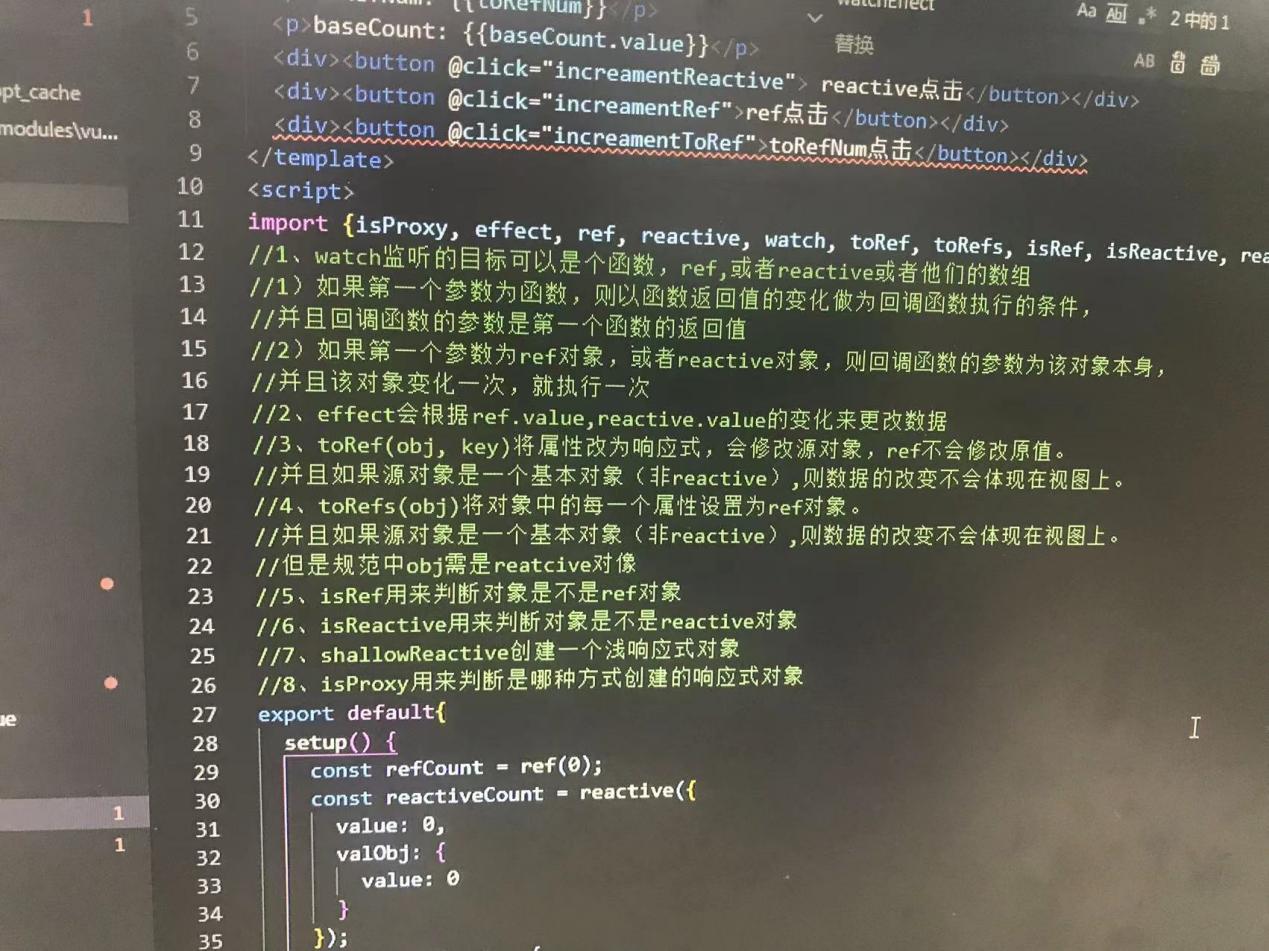
console.log(data.aa)

19、setup，effetc,ref,reactive等的实现原理

以及vue3的源码解析，见D盘爪哇/vue/vue3的源码与实战中的reactive.esm-broswer.js

与vue.esm-broswer.js文件。

20、补充知识



# 8、vue3的生命周期

1）setup语法糖模式没有beforeCreate和created这两种生命周期的，setup 去代替，

然后执行onBeforeMount(() => {}),再执行onMounted(() => {})

onBeforeUpdate(() => { //获取更新之前的DOM

})

onUpdated(() => { //获取的是更新之后的DOM

})

onBeforeUnMount(() => {})

onUnmounted(() => {})

onRenderTracted(() => {}) //调试用，收集依赖的时候触发

onRenderTriggered(() => {}) //调试用，更新依赖的时候触发

# 9、父子组件传值

1）父传子略defineProps<Props>(),设置默认值：wiithDefaults(defineProps<>(), {name: “默认名称”,arr: () => [1, 2, 3, 4]}) //复杂数据类型需要采用函数返回值的写法。

2）子传父：子组件const emit = defineEmits([“selfEvent”]) cliclToChangeVal = () => {emit(“selfEvent”, params)}

3）父组件通过ref获取子元素以及子元素上面的属性和方法

父组件引入的子组件上：ref=”childTemplate”(虚拟DOM上)

父组件script中

const childTemplate= ref(null)

子组件script中

defineExpose({ //将需要在父组件获取的属性和方法传出去

data

})

# 10、 全局组件，局部组件和递归组件

1）全局组件注册

mian.ts中。

import Card from “../../component/Card.vue”

createApp(App).component(“组件名称，可以随意定义”,Card ).mount(“#app”)

注册完之后在任何模块就可以直接调用了

2）局部组件略

3）递归组件

给需要递归的组件定义一个name，然后在组件中渲染（加上递归结束的条件）

# 11、动态组件

<component :is=”currentTemplate”></component>