学习资料

名称	链接	说明
vue3.0	https://v3.vuejs.org	英文文档
vue3.0	https://vue3js.cn/docs/zh/	中文文档
GitHub(vue源码)	https://github.com/vuejs/vue-next (3.0) https://github.com/vue (2.0)	vue github源码
vue相关插件源码	https://github.com/vuejs/vue-router-next https://github.com/vuejs/vuex https://github.com/element-plus/element-plus	源码

相关资料

源码：https://github.com/vuejs/vue-next (3.0源码)

介绍：https://www.bilibili.com/video/BV1Tg4y1z7FH

模板：https://vue-next-template-explore

模板：https://vue-next-template-explorer.netlify.app/

RFC： https://composition-api.vuejs.org/

Vue3.0重要变化

1. 更快的性能
2. 组合API(最大的亮点 setup)
3. Fragment teleprot subspense
4. 自定义渲染API
5. 更好的支持TypeScript
6. 支持tree-shaking

为什么Vue3.0更快的性能

1. **diff算法 (附录1) （https://vue-next-template-explorer.netlify.app/）**

   2.0 全量比较DOM

3.0 新增加了静态标记（PatchFlag）

在与上次虚拟节点进行比较的时候，只对比带patch flag的节点

并且可以通过flag的信息得知当前节点要对比的具体内容

```
附录1.diff只会对msg的进行对比

//这里的flag就是1 变化的只有text文本
<div>
 <p>你好</p>
 <p>你好</p>
 <p>你好</p>
 <p>{{msg}}</p>
</div>
 export function render(_ctx, _cache, $props, $setup, $data, $options) {
  return (_openBlock(), _createBlock("div", null, [
  _createVNode("p", null, "你好"),
  _createVNode("p", null, "你好"),
  _createVNode("p", null, "你好"),
  _createVNode("p", null, _toDisplayString(_ctx.msg), 1 /* TEXT */)
  ]))
 }


 // 这里的flag值就是3 变化的有text和class
 <div>
  <p>你好</p>
  <p>你好</p>
  <p>你好</p>
  <p>{{msg}}</p>
  <p :class="{'isNew': msg='123'}">{{msg}}</p>
 </div>


 export function render(_ctx, _cache, $props, $setup, $data, $options) {
  return (_openBlock(), _createBlock("div", null, [
  _createVNode("p", null, "你好"),
  _createVNode("p", null, "你好"),
  _createVNode("p", null, "你好"),
  _createVNode("p", null, _toDisplayString(_ctx.msg), 1 /* TEXT */),
  _createVNode("p", {
  class: {'isNew': _ctx.msg='123'}
```

```
  }, _toDisplayString(_ctx.msg), 3 /* TEXT, CLASS */)
  ]))
 }
```

2. hostStatic 静态提升（附录2）

2.0中无论元素是否参与更新，每次都会重新创建，然后就渲染

3.0 对不参与更新的元素，会做静态提升，只会创建一次，在渲染的时候复用

```
附录2
//vue2.0 每次都会createVnode vue3.0会把不需要的缓存下来
const _hoisted_1 = /*#__PURE__*/_createVNode("p", null, "你好", -1 /* HOISTED */)
const _hoisted_2 = /*#__PURE__*/_createVNode("p", null, "你好", -1 /* HOISTED */)
const _hoisted_3 = /*#__PURE__*/_createVNode("p", null, "你好", -1 /* HOISTED */)

export function render(_ctx, _cache, $props, $setup, $data, $options) {
 return (_openBlock(), _createBlock("div", null, [
 _hoisted_1,
  _hoisted_2,
  _hoisted_3,
  _createVNode("p", null, _toDisplayString(_ctx.msg), 1 /* TEXT */),
  _createVNode("p", {
  class: {'isNew': _ctx.msg='123'}
  }, _toDisplayString(_ctx.msg), 3 /* TEXT, CLASS */)
  ]))
 }
```

3. cacheHandlers 事件侦听器缓存 （附录3）

默认情况下onClick会被视为动态绑定，所以每次都会去追踪它的变化

但是因为是同一个函数，所以没有追踪变化，直接缓存起来复用即可

```
附录3
<div>
 <div @click="onClick"></div>
</div>
// 没有开启缓存
export function render(_ctx, _cache, $props, $setup, $data, $options) {
 return (_openBlock(), _createBlock("div", null, [
  _createVNode("div", { onClick: _ctx.onClick }, null, 8 /* PROPS */, ["onClick"])
```

```
9   ]))
10  }
11
12  // 开启之后
13  export function render(_ctx, _cache, $props, $setup, $data, $options) {
14   return (_openBlock(), _createBlock("div", null, [
15   _createVNode("div", {
16   onClick: _cache[1] || (_cache[1] = (...args) =>
(_ctx.onClick(...args)))
17   })
18   ]))
19  }
```

4.ssr

当有大量静态内容时，这些内容会被当做纯字符串推进一个buffer里面

即使存在动态的绑定，会通过模板插值嵌入进去，这样会比通过虚拟Dom渲染快很多

当静态内容大到一定量级的时候，会用createStaticVNode方法在客户端去生成一个static node 这些node,会被直接innerHtml，不需要创建对象，然后根据对象渲染

Vue3.0项目搭建

第一种：Vue-cli方式

- 下载最新的vue-cli脚手架
- winpty vue.cmd create project-name(这里是使用winpty + vue.cmd 命令创建项目，可切换的选项，初始化安装 vue-router vuex css less 等等必备依赖包)
- vue3.0 需要使用vue-next插件升级的方式，目前vue-cli还不支持vue3.0，输入指令 vue add vue-next
- 开始vue3.0撸码

第二种：webpack方式

- git clone https://github.com/vuejs/vue-next-webpack-preview.git
- cd projectName
- npm install
- npm run dev

第三种:Vite方式(比webpack更快，vue团队的脚手架)

- npm install -g create-vite-app
- create-vite-app projectname
- npm install (yarn install)
- npm run dev (yarn run dev)

Vue 3.0 Composition API

vue2.0存在的问题--(业务逻辑分散，数据和处理办法都被 methods watch computed等分隔开)

```
// 数据和处理办法都被 methods watch computed等分隔开
export default {
 data () {
 return {
 // 购物车相关的数据定义
 shopCartList: [],
 ...

 // 用户相关
  user: {}
  ...
 }
 },
 methods: {
 // 购物车逻辑
 addShopCart () {},
 deleteShopCart () {},

 // 用户相关的逻辑
 checkUser () {},
 loginOutUser () {}
 },
 watch: {
 // 购物车相关
 shopCartList () {},

```

```
  // 用户相关的逻辑
  user () {},
  },
  computed: {
  // 购物车相关
  shopCartList () {},

  // 用户相关的逻辑
  user () {},
  }
}
```

vue3.0组合api横空出世

setup()

最大亮点就是setup(),setup是组合api的入口函数.

step函数接收两个参数,一个是props属性，可以拿到组件props属性中的数据。第二个参数是context，是一个普通的JavaScript对象。

props

props是响应式的，不能使用ES6解构，会消除props属性。如果想使用props，可以通过toRefs(props)进行解构。解构后直接任意修改，全是响应式的(不知这个bug后期是否会修复，完全不符合vue的数据单向流)

context

是一个普通的JavaScript对象，它暴露三个组件的property

context.attrs(Attribute 非响应式对象属性)

context.solts(插槽 非响应式对象)

context.emit(触发事件 方法)

因为它是非响应的，可以放心的使用ES6解构。

context.emit用的比较频繁，通过context.emit触发事件，传递给子组件。如下图

父组件

```
<template>
 <div id="nav">
 <p> {{`这是传递给子组件中的数据:${name}`}}</p>
 // 传递的value传递给子组件 通过change来接收子组件传递给来的数据
 <router-view :value ='name' @change="updateData"/>
 </div>
</template>
<script>
 import {ref} from 'vue'
```

```
 export default {
 name:'app',
 setup(){
 const name = ref('你好')
 const updateData =(e)=>{
 name.value = e
 }
 return {
 name,
 updateData
 }
 }
 }
</script>
```

子组件

```
export default {
 name: "Home",
 //注意这里要加上emits,
 //把触发父组件的方法暴露出来，否则会报警告
 emits:['change'],
 props:{
 value:{
 type:String,
 default:''
 }
 },
 setup(props,context){
 const add = ()=>{
 context.emit('change','20')
 }
 return {
 add
 }
 }
};
```

注意：

1. setup()是在beforeCreated和created之间 组件的data，methods还未初始化，所以在setup中无法使用data和methods。

2. setup函数只能是同步的
3. setup函数中this为undefined

响应式数据两种方式(就是2.x中data(){return{}})
第一种带ref的响应式变量
第二中是reactive方式的响应式变量
setup中使用ref,reactive,isRef, isReactive,shallowRef,shallowReactive, triggerRef, toRow, markRaw，toRef, toRefs， customRef， readonly ，shallowReadobly , isReadonly等概念

1.reactive是什么？监听复杂类型变量
是vue3.0提供得实现响应式数据得方法
在vue2.0的时候是通过Obejct.definedProperty来实现
在vue3.0的时候是通过ES6的Proxy来实现的
2.reactive注意点
参数必须是对象(JSON/arr)
默认情况下修改对象不会触发界面更新
想更新就必须重新赋值

3. ref 监听简单类型
ref和reactive一样都是实现响应式数据的方法
reactive必须传一个对象，所以一个变量想实现响应式就比较麻烦，所以ref帮助我们实现变量的响应式
ref本质还是一个reactive，系统会自动根据我们给ref传入的值将他转换
ref(xx)=>reactive({value:xxx})
4. ref 注意点
在vue模板中使用ref的值不需要value获取
js中使用或者赋值必须使用ref.value

5.isRef，isReactive判断是不是ref或者isReactive创建的

6.shallowRef, shallowReactive 非递归监听
- ref 和 reactive 都是递监听的 存在的问题是性能消耗严重（每一层都会包装成proxy）所以可以使用shallow + xx 来表示非递归监听的方法
- * 通过shallowReactve创建的数据，只会监听第一层的变化
- * 通过shallowRef创建的数据，那么Vue监听的是.value的变化，而不是第一层的变化
- * 使用场景： 一般使用ref和reactive就可以，只有数据量比较大使用shallowRef和shallowReactive

7. shallowRef本质
   ref 本质
   ref(xx) -> reactive({value:xx})
   shallowRef本质
   shallowRef(xx) -> shallowReactive({value:xx})
   * 通过shallowRef创建的数据，那么Vue监听的是.value的变化

8. triggerRef会根据传入的值，更新一遍数据
   - * v3只提供了triggerRef方法，并没有提供triggerReactive方法。
     那么reactive类型的数据，无法主动触发界面更新

9. toRow是什么
   用来获取ref或者reactive的原始数据

```
let obj = {name: 'jack', age: 18}
let state = reactive(obj)
let obj2 = toRaw(state)
obj2 === obj
```

   如果是ref定义的 想通过toRaw处理得到原始值 要获取的是 。value的值 因为经过vue处理之后， .value中保存的才是当初创建是传入的那个原始数据

```
let count = 1
let state = ref(count)
let count2 = toRaw(count.value)
count === count2
```

10. markRaw是什么
   如果你希望一个数据永远都不要被监听就可以使用markRaw

```
let obj = {name: 'jack', age: 18}
let obj = markRaw(obj)
let state = reactive(obj)
```

   修改state的值将不会引发页面相应，因为MarkRaw已经将obj变成永远无法监听的响应式数据

11. toRef是什么
   想对一个对象中的某个属性进行响应式处理

```
let obj = { name: 'jack'}
let status = toRef(obj, 'name')
const myFun = () => {
   status.value = 'jobs'
```

```
        console.log(obj) // {name: "jobs"}
        console.log(status)
        // ObjectRefImpl {
        //  __v_isRef: true
        //  _key: "name"
        //  _object: {name: "jobs"}
        //  value: "jobs"
        // }
    }
```

ref和toRef区别

ref本质是复制，修改响应式数据不会影响以前的数据

toRef本质是引用，修改响应式数据的值是会影响以前的数据

ref数据变化，界面就会自动更新

toRef数据变化，界面不会自动更新

应用场景

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

```
let obj = {name: 'jack', age: 18}
let name = toRef(obj, 'name')
```

12. toRefs是什么

想把某一个对象中的多个属性变成响应式对象

```
let obj = { name: 'jack', age: 18}
let status = toRefs(obj)
const myFun = () => {
    status.name.value = 'jobs'
    status.age.value = 20
    console.log(obj) // {name: "jobs", age: 20}
    console.log(status) // 附件4
}
```

13. customRef是什么

自定义ref方法

```
function myRef (val) {
  return customRef((track, trigger) => {
    return {
      get () {
        track() // 告诉我们Vue 这个数据是需要追踪变化
        console.log('get', val)
        return val
```

```
      },
      set (newValue) {
        console.log('set', newValue)
        val = newValue
        trigger() // 告诉我们Vue 触发我们界面更新
      }
    }
  })
}
```

14. 为什么要使用customRef

serup中不能使用async 但是我们确实想在其中使用就可以将异步代码写在自定义的方法里面

```
let  age = myRef('../public/data.json')
function myRef (val) {
    return customRef((track, trigger) => {
        fetch(val).then(res => {
          return res.json()
        }).then(data => {
          console.log(data)
          val = data
          trigger()
        })
        return {
          get () {
            track() // 告诉我们Vue 这个数据是需要追踪变化
            console.log('get', val)
            return val
          },
          set (newValue) {
            console.log('set', newValue)
            val = newValue
            trigger() // 告诉我们Vue 触发我们界面更新
          }
        }
    })
}
```

15. ref 与 原来2.0 ref有什么区别

在setup中不能使用$refs, 那我们如何使用ref标记的DOM

```
setup () {
  const box = ref(null)
```

```
    onMounted(() => {
        console.log('onMounted', box.value)
    })
    console.log(box.value)
    return {
      box
    }
  }
```

16. setup中需要使用生命周期 从vue中引入， 上述例子中使用了 onMounted

```
import { ref, onMounted } from 'vue'
setup () {
    const box = ref(null)
    onMounted(() => {
      console.log('onMounted', box.value)
    })
    return {
        box
    }
}
```

17. readonly

通过readonly创建只读的数据，而且是递归只读!

18 shallowReadobly

通过shallowReadobly创建只有第一层是只读的数据， 不是递归只读!

19. isReadonly

判断 是不是通过readonly创建只读的数据

生命周期钩子注册内部setup

组合式 API 上的生命周期钩子与选项式 API 的名称相同，只是前缀添加on，及onMouted对应2.x的
mouted。*其他的*
onBeforeMount,onMounted,onBeforeUpdate,onUpdated,onBeforeUnmount,
onUnmounted 相关生命周期都可以注册到内部setup

例如 。各个生命周期函数的参数都是hook钩子函数，这个和2.x created(){} mounted(){}直接执行代码块还是有一些区别的 ， 选项式API和Hook inside的区别

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`
`errorCaptured`	`onErrorCaptured`
`renderTracked`	`onRenderTracked`
`renderTriggered`	`onRenderTriggered`

```
setup(props,context){
 const a = {
 a:1,
 list:[],
 dataArr:[2,3,4]
 }
 const state = reactive(a)

 let app = ref(11)
  //各个生命周期内部都是一个hook钩子函数
  onMounted(async ()=>{
  await LIST().then(res=>{
  if(res.status === 200){
  state.list = res.data
  }else {
  alert('有错误')
  }

  })

  })
```

```
  onBeforeMount(()=>{
  console.log('onBeforeMount')
  })
  onBeforeUpdate(()=>{
  console.log('onBeforeUpdate')
  })
  onUpdated(()=>{
  console.log('onUpdated')
  })
  onBeforeUnmount(()=>{
  console.log('onBeforeUnmount')
  })
  onUnmounted(()=>{
  console.log('onUnmounted')
  })

  return {
  state,
  app
  }
}
```

watch响应式更改

就像我们如何使用2.x watch选项在组件内的user property上设置监听器一样，我们也可以使用Vue导入watch函数执行相同的操作。它接受3个参数。一个是响应式引用或者想要监听的getter函数，一个是回调，一个是可选的配置选项。

```
 import { ref, watch } from 'vue'
const counter = ref(0)
// watch的第一个参数是一个响应式的引用 第二个参数是回调 第三个参数是一个可选配置
watch(counter, (newValue, oldValue) => {
 console.log('The new counter value is: ' + counter.value)
})
```

watch应用

```
// src/components/UserRepositories.vue `setup` function
import { fetchUserRepositories } from '@/api/repositories'
import { ref, onMounted, watch, toRefs } from 'vue'
// 在我们组件中
```

```
setup (props) {
 // 使用 `toRefs` 创建对prop的 `user` property 的响应式引用
 const { user } = toRefs(props)
 const repositories = ref([])
 const getUserRepositories = async () => {
 // 更新 `prop.user` 到 `user.value` 访问引用值
 repositories.value = await fetchUserRepositories(user.value)
 }
 onMounted(getUserRepositories)
 // 在用户 prop 的响应式引用上设置一个侦听器
 watch(user, getUserRepositories)
 return {
 repositories,
 getUserRepositories
 }
}
```

独立的computed属性

与ref何watch类似,也可以使用Vue导入computed函数在Vue组件外部创建计算属性，让我们回到我们的counter例子：

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

const counter = ref(0)
const twiceTheCounter = computed(() => counter.value * 2)

counter.value++
console.log(counter.value) // 1
console.log(twiceTheCounter.value) // 2
```

通过把一些嵌合的组件抽出来，分成多个功能文件。

提供/注入

我们也可以在组合API中使用provide/inject(可注入多种数据，方法)． Vue2.X的使用方法父组件提供

```
export default {
 provide() {
 return {
 reactive: this.reactive
```

```
};
},

data() {
return {
 reactive: {
 value: "Hello there"
 }
 };
 }
};
```

子组件注入

```
inject:['reactive']
```

3.0在setup中的使用方法

父组件提供

```
<script>
import { provide,readonly} from 'vue'
import MyMarker from './MyMarker.vue
//如果要确保通过 provide 传递的数据不会被注入的组件更改,
//我们建议对提供者的 property 使用 readonly
export default {
 components: {
 MyMarker
 },
 setup() {
 const location = ref('North Pole')
 const geolocation = reactive({
 longitude: 90,
 latitude: 135
 })
 //建议尽可能，在提供者内保持响应式 property 的任何更改,这里可以
 // 内部提供一个一个方法负责改变响应式 property。
 const updateLocation = () => {
 location.value = 'South Pole'
 }
 provide('location ', readonly(location))
 provide('geolocation ', readonly(geolocation))
```

```
 provide('updateLocation', updateLocation)
 }
}
</script>
```

子组件注入使用

```
<script>
import { inject} from 'vue'
export default {
 setup() {
 const userLocation = inject('location', 'The Universe')
 const userGeolocation = inject('geolocation')
 const updateLocation = inject('updateLocation')
 return {
 userLocation,
  userGeolocation,
  updateLocation
 }
 }
}
</script>
```

模板引用

在使用组合式 API 时，响应式引用和模板引用的概念是统一的。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声明 ref 并从 setup() 返回

```
<template>
 <div ref="root">This is a root element</div>
</template>
<script>
 import { ref, onMounted } from 'vue'
 export default {
 setup() {
 const root = ref(null)
 onMounted(() => {
  // DOM元素将在初始渲染后分配给ref
  console.log(root.value) // <div>这是根元素</div>
 })

 return {
```

```
 root
 }
 }
 }
</script>
```

jsx 语法使用ref，获取当前ref的模板元素

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

 return () =>
 h('div', {
 ref: root
 })

 // with JSX
 return () => <div ref={root} />
 }
}
```

Vue3.0使用变化总结

1. 初始化Vue

```
//2.x版本
new Vue({
 el: '#root'
})

//3.0是直接使用createApp，接收vue
import { createApp } from "vue";
import App from "./App.vue";
import router from "./router";
import store from "./store";
//3.0 是直接使用createApp 挂载根组件 APP
const app = createApp(App)
//3.0 直接使用 use 然后再链式注入 store 和 router
app.use(store).use(router).mount("#app");
```

2. router.js 路由的引入

```
//3.0 使用 createRouter创建router实例,
//调用createWebHashHistory函数为hash模式
//调用createWebHistory函数为history模式
import { createRouter, createWebHashHistory } from "vue-router";
import Home from "../views/Home.vue";
const routes = [
 {
 path: "/",
 name: "Home",
  component: Home,
  },
  {
  path: "/about",
  name: "About",
  // route level code-splitting
  // this generates a separate chunk (about.[hash].js) for this route
  // which is lazy-loaded when the route is visited.
  component: () =>
  import(/* webpackChunkName: "about" */ "../views/About.vue"),
  },
 ];
 //3.0 使用 createRouter创建router实例
 const router = createRouter({
  history: createWebHashHistory(), // createWebHashHistory() hash模式 // c
reateWebHistory() history模式
  routes,
 });

 export default router;
```

3.vuex引入,store文件

```
// 3.0使用createStore接收vuex
import { createStore } from "vuex";
const options = {
 state: {},
 mutations: {},
 actions: {},
 modules: {},
```

```
}
// 通过createStore导出options options为一个对象
//里面包含之前的一样state mutations actions 和 modules
export default createStore(options);
```

4.Filters 已从 Vue 3.0 中删除，不再受支持

5.v-if和v-for的优先级对比

2.x 版本中在一个元素上同时使用 v-if 和 v-for 时，v-for 会优先作用

3.x 版本中 v-if 总是优先于 v-for 生效。

由于语法上存在歧义，建议避免在同一元素上同时使用两者

比起在模板层面管理相关逻辑，更好的办法是通过创建计算属性筛选出列表，并以此创建可见元素

6. 多事件处理

```
只有3.0有 多事件并不是每次执行一个，而是依次执行的
<!-- 这两个 one() 和 two() 将执行按钮点击事件 -->
<button @click="one($event), two($event)">
 Submit
</button>

one(event) {
 // first handler logic...
 console.log('one')
  this.counter++
 },
two(event) {
  // second handler logic...
  console.log('two')
  this.counter++
 },
three(event) {
  // second handler logic...
  console.log('three')
  this.counter++
 }
```

7. Vue 3 中，组件现在正式支持多根节点组件

8.Vue3中， 可以只用teleport 在body下面创建元素

vue2.x版本如果需要添加自定义指令的话。原始挂载的父节点移除，然后挂载到body上去，通过手动的形式来重新挂载，能够很好的解决这种问题，当然上面只是简单的逻辑，如果需要考虑卸载等其他逻辑代码还得增加。

vue3.0 Teleport能够直接帮助我们将组件渲染后页面中的任意地方，只要我们指定了渲染的目标对象。Teleport使用起来非常简单。

```
1
2  <template>
3    <teleport to="body" class="modal" v-if="show">
4      <div class="modal-mask" @click="close"></div>
5      <slot></slot>
6    </teleport>
7  </template>
8
9  <script>
10  import "./style.scss";
11  export default {
12    props: {
13      show: Boolean,
14    },
15    methods: {
16      close() {
17        this.$emit("close");
18      },
19    },
20  };
21  </script>
```

9.Render 函数参数

```
1  // Vue 2 渲染函数示例
2  export default {
3   render(h) {
4   return h('div')
5   }
6  }
7  h 是 createElement 别名
8
9  // Vue 3 渲染函数示例
10  import { h } from 'vue'
11  export default {
12   render() {
13   return h('div')
```

```
14    }
15  }
```