# Vue响应式原理

## Vue2.0 响应式原理

### 原理：

当你把一个普通的JavaScript对象传入vue实例作为data选项，Vue将遍历此对象所有的property，并使用Object.defineProperty 把这些property全部转为Getter/setter。

Object.defineProperty 是Es5 中一个无法shim 的特性，这也是Vue不支持IE8 以及更低版本浏览器的原因。

每一个Vue实例都有一个watch 实例，它会在实例渲染时记录这些属性，并在setter触发时重新渲染。

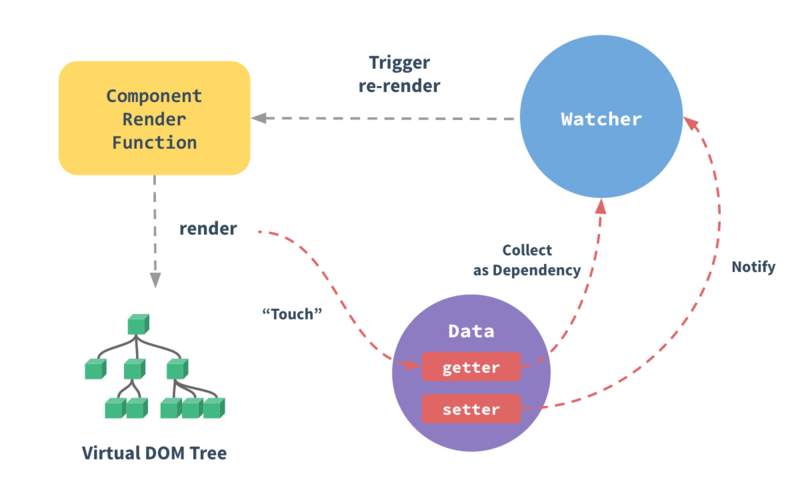
### Watch

什么是Watch实例？ 每一个组件在渲染之前，会对其自身的template模板进行解析编译，生成一个render函数（用于创建虚拟dom）。然后，会实例化一个watch对象，在实例化过程中，会将之前生成的render函数函数作为参数传入进去执行。而在执行render的过程中，势必触发get函数，将这个watch实例添加到相应属性的订阅数组subs当中。等这些属性进行被修改时，就会通知这个watch，再次调用这个watch里面的render函数，进行虚拟dom的diff和更新

### 缺点：

1. Vue无法检测到对象属性的添加或者删除
2. Vue不支持动态添加根节点级别的响应式属性。但是，可以使用Vue.set(object,PropertyName,Value) 方法嵌套对象添加响应式属性。

3、据的响应式处理和视图未完全解耦

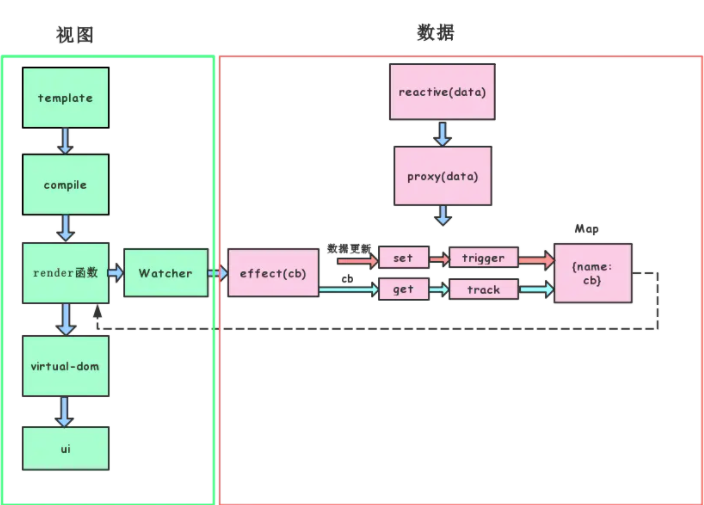


### 总结：

所谓的响应式处理方式，是拦截属性的get/set 方法做订阅-发布处理。订阅-发布，就是在属性被调用的使用，触发get代理函数，订阅用该属性的组件（将组件存放到一个订阅者数组中进行保存，这里的Dep.target暂时可以理解为一个全部变量，代表着的是正在渲染的Vue组件）。而在属性被修改时，触发set 代理函数，在set代理函数里，通过订阅数组里面的每一个订阅者（组件）进行试图更新。

## Vue3.0 响应式原理

### Vue3.0响应式原理框架图



由图中我们看出，Vue3.0响应式框架在设计上，将视图渲染和数据响应式完全分离开来。将响应式核心方法effect从原有的Watcher中抽离。这样，当我们只需要监听数据响应某种逻辑回调(例如监听某个text属性的变化，对他进行正则校验等。)，而不需要更新视图的时候，完全可以只从effect触发，屏蔽左边的视图渲染。

### Vue3.0原理

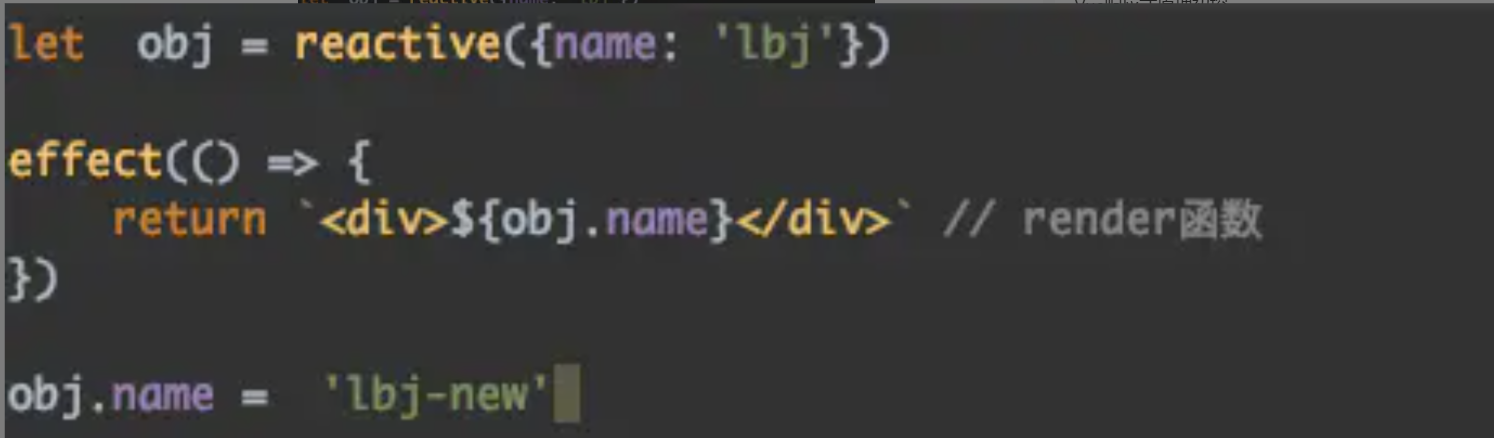
Vue3.0响应式相对于Vue2.0的最大区别在于用到了es6中的方法Proxy。这个方法不需要循环遍历data的每个属性，对每个属性都做一遍响应式处理。而是直接代理了整个data对象，拦截这个对象所包含的所有属性的get、set方法。这么做的好处:

1.就是在我们动态为data添加一个属性时，不用做任何处理，这个属性就已经是响应式的了。

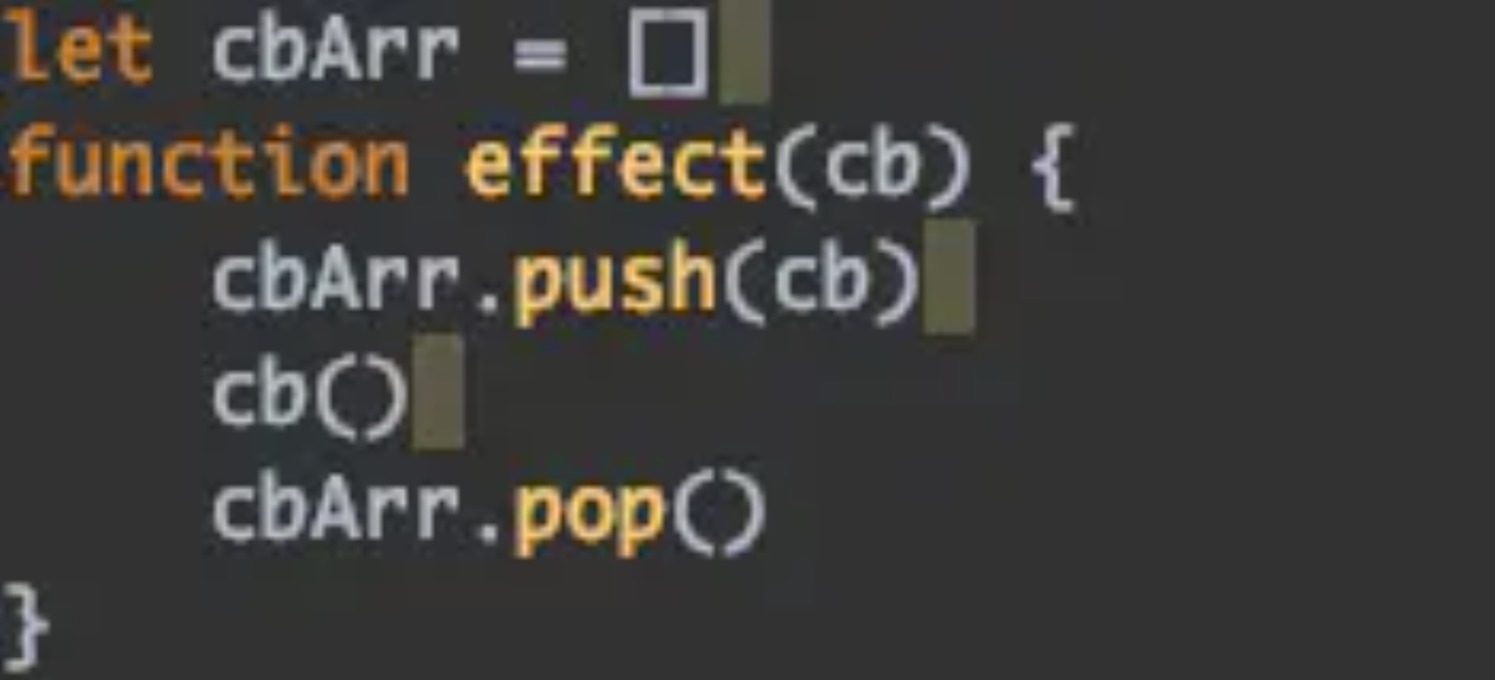
2.数组的任何操作也都可以触发响应。

### Effect

上面的设计，其实已经解决了Vue2.0的前两个不足。而为了解决第三个不足， Vue3.0提出了effect函数，其用法如下：



当我们调用effect执行某个函数时，首先会执行一次这个函数。之后每次obj.name修改时，执行一次该函数。那么effect是怎么实现这个功能的？



effect会先将传入的回调函数保存起来，然后第一次执行cb(),在执行的过程中，触发了obj.name属性的get代理函数，在代理函数里面做订阅track(target), 而track函数则会将存在全局数组中的cb函数取出来，保存在map中,类似于map.set('name',cb)。之后，在name属性被修改的时候，会触发name属性的set代理函数，在代理函数里面执行trigger。最后，trigger会将之前存在map里的订阅着取出来执行，map.get('name')()

### proxy

#### proxy前景

vue2.x 使用Object.defineProperty 作为响应式原理实现，但是会有它的局限性：

1. 无法监听数组基于下标的修改
2. 不支持Map 、 Set 、 WeakMap 和 WeakSet

改用了 proxy解决了这些问题，这也意味着vue3.0 将放弃对低版本浏览器的兼容（兼容IE11以上版本）

#### Proxy中hander对象的基础用法

**Vue3.0 响应式用到的捕获器：**

* **Handler.has() -> in 操作符的捕捉器**
* **Handler.get() -> 属性读取操作的捕捉器**
* **Handler.set() -> 属性设置操作的捕捉器**
* **Handler.deleteProperty() -> delete操作符的捕捉器**
* **Handler.ownKeys() -> Object.getOwnPropertyNames 方法 和**

**Object.getOwnPropertySymbols 方法的捕捉器**

##### **Has捕捉器**

has(targer, propKey)

target：目标对象

propKey：待拦截属性名

作用：拦截判断target对象是否含有属性propKey的操作

拦截操作：propKey in proxy 不包含for…in 循环

对应Reflect：Reflect.has(target, propKey)

实例：

**const** handler **=** {

has(target, propKey){

*/\**

*\* 做你的操作*

*\*/*

**return** propKey **in** target

}

}

**const** proxy **=** **new** Proxy(target, handler)

##### get捕捉器

get(target, propKey, receiver)

target: 目标对象

propkey: 待拦截的属性名

receiver：proxy实例

返回：返回读取的属性

作用：拦截对象属性的读取

拦截操作：proxy[propKey] 或者点运算符

对应Reflect：Reflect.get(target, propertyKey[,receiver])

例子

**const** handler **=** {

get**:** **function**(obj, prop) {

**return** prop **in** obj **?** obj[prop] **:** '没有此水果';

}

}

**const** foot **=** **new** Proxy({}, handler)

foot.apple **=** '苹果'

foot.banana **=** '香蕉';

console.log(foot.apple, foot.banana); */\* 苹果 香蕉 \*/*

console.log('pig' **in** foot, foot.pig); */\* false 没有此水果 \*/*

特殊情况：

**const** person **=** {};

Object.defineProperty(person, 'age', {

value**:** 18,

writable**:** **false**,

configurable**:** **false**

})

**const** proxPerson **=** **new** Proxy(person, {

get(target,propKey) {

**return** 20

*//应该return 18;不能返回其他值，否则报错*

}

})

console.log( proxPerson.age ) */\* 会报错 \*/*

##### set 捕捉器

set(target,propKey, value,receiver)

target:目标对象

propKey:待拦截属性名

value:新设置的属性值

receiver: proxy实例

返回：严格模式下返回true操作成功；否则失败，报错

作用： 拦截对象的属性赋值操作

拦截操作： proxy[propkey] = value

对应Reflect： Reflect.set(obj, prop, value, receiver)

**let** validator **=** {

set**:** **function**(obj, prop, value) {

**if** (prop **===** 'age') {

**if** (**!**Number.isInteger(value)) { */\* 如果年龄不是整数 \*/*

**throw** **new** TypeError('The age is not an integer')

}

**if** (value **>** 200) { */\* 超出正常的年龄范围 \*/*

**throw** **new** RangeError('The age seems invalid')

}

}

obj[prop] **=** value

*// 表示成功*

**return** **true**

}

}

**let** person **=** **new** Proxy({}, validator)

person.age **=** 100

console.log(person.age) *// 100*

person.age **=** 'young' *// 抛出异常: Uncaught TypeError: The age is not an integer*

person.age **=** 300 *// 抛出异常: Uncaught RangeError: The age seems invalid````*

**注意：**当对象的属性writable为false时，该属性不能在拦截器中被修改

**const** person **=** {};

Object.defineProperty(person, 'age', {

value**:** 18,

writable**:** **false**,

configurable**:** **true**,

});

**const** handler **=** {

set**:** **function**(obj, prop, value, receiver) {

**return** Reflect.set(...arguments);

},

};

**const** proxy **=** **new** Proxy(person, handler);

proxy.age **=** 20;

console.log(person) *// {age: 18} 说明修改失败*

##### **deleteProperty捕获器**

deleteProperty(target, propKey)

target:目标对象

propKey:待拦截属性名

返回：严格模式下只有返回true, 否则报错

作用： 拦截删除target对象的propKey属性的操作

拦截操作： delete proxy[propKey]

对应Reflect： Reflect.delete(obj, prop)

**var** foot **=** { apple**:** '苹果' , banana**:**'香蕉' }

**var** proxy **=** **new** Proxy(foot, {

deleteProperty(target, prop) {

console.log('当前删除水果 :',target[prop])

**return** **delete** target[prop]

}

});

**delete** proxy.apple

console.log(foot)

*/\**

*运行结果：*

*'当前删除水果 : 苹果'*

*{ banana:'香蕉' }*

*\*/*

特殊情况： 属性是不可配置属性时，不能删除

**var** foot **=** { apple**:** '苹果' }

Object.defineProperty(foot, 'banana', {

value**:** '香蕉',

configurable**:** **false**

})

**var** proxy **=** **new** Proxy(foot, {

deleteProperty(target, prop) {

**return** **delete** target[prop];

}

})

**delete** proxy.banana */\* 没有效果 \*/*

console.log(foot)

##### ownKeys捕捉器

ownKeys(target)

target：目标对象

返回： 数组（数组元素必须是字符或者Symbol,其他类型报错）

作用： 拦截获取键值的操作

拦截操作：

1 Object.getOwnPropertyNames(proxy)

2 Object.getOwnPropertySymbols(proxy)

3 Object.keys(proxy)

4 for...in...循环

对应Reflect：Reflect.ownKeys()

**var** obj **=** { a**:** 10, [Symbol.**for**('foo')]**:** 2 };

Object.defineProperty(obj, 'c', {

value**:** 3,

enumerable**:** **false**

})

**var** p **=** **new** Proxy(obj, {

ownKeys(target) {

**return** [...Reflect.ownKeys(target), 'b', Symbol.**for**('bar')]

}

})

**const** keys **=** Object.keys(p) *// ['a']*

*// 自动过滤掉Symbol/非自身/不可遍历的属性*

*/\* 和Object.keys()过滤性质一样，只返回target本身的可遍历属性 \*/*

**for**(**let** prop **in** p) {

console.log('prop-',prop) */\* prop-a \*/*

}

*/\* 只返回拦截器返回的非Symbol的属性，不管是不是target上的属性 \*/*

**const** ownNames **=** Object.getOwnPropertyNames(p) */\* ['a', 'c', 'b'] \*/*

*/\* 只返回拦截器返回的Symbol的属性，不管是不是target上的属性\*/*

**const** ownSymbols **=** Object.getOwnPropertySymbols(p)*// [Symbol(foo), Symbol(bar)]*

*/\*返回拦截器返回的所有值\*/*

**const** ownKeys **=** Reflect.ownKeys(p)

*// ['a','c',Symbol(foo),'b',Symbol(bar)]*

#### 建立响应式

Vue3.0 建立响应式的方法有两种：

1. 运用composition-api中的reactive直接构建响应式，composition-api的出现我们可以在

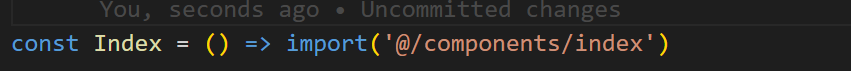
.vue 文件中，直接使用setUp() 函数处理之前的大部分逻辑，也就是说我们没有必要的export default{} 中在声明生命周期，data(){} 函数，watch{}，computed{}等，取而代之的是我们在setup函数中，用vue3.0 reactive watch 生命周期api 来达到同样的效果，这样就像 react-hooks一样提升代码的复用率， 逻辑性更强

2、用传统的data(){} 形式，vue3.0 没有放弃对vue2.0 写法的支持，而是对vue2.0 的写法是完全兼容的，提供了applyOptions来出来options形式的vue组件。 但是options里面的data，watch，computed等处理逻辑 ，还是用了composition-api中的API 对应处理

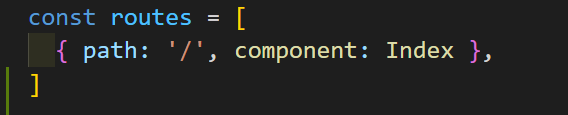
# Vue2.X

## 路由懒加载配置

1. 组件导入方式，直接使用异步组件的导入方法



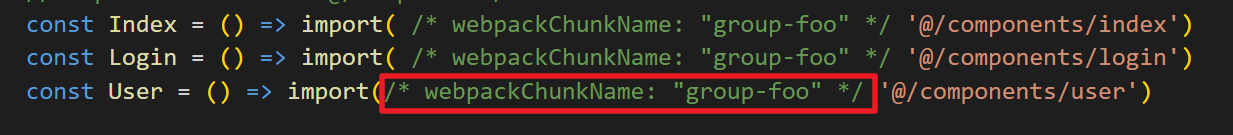
1. 路由配置



1. 异步路由网络请求时机

当路由访问到的时候，才会加载响应的页面的资源

把对应异步路由打包到同一个资源文件，需要通过特殊的注释标记



## Vue项目热启动配置

### webpack Vue

#### 需要的依赖包：

vue

vue-loader

vue-template-compiler

webpack

babel-loader babel-core babel-preset-2015 babel-preset-stage-2

css-loader

#### 配置文件

const path = require('path')

const webpack = require('webpack')

const HtmlWebpackPlugin = require('html-webpack-plugin')

const resole = (dir)=> path.resolve(\_\_dirname, '../' + dir)

module.exports = {

//配置文件入口

entry:{

app:'./src/main.js'

},

//配置输出目录

output:{

path:resole('dist'),

filename:'[name].[hash].js'

},

//配置编译需要的加载器

module:{

rules:[

{

test: /\.vue$/,

use:[

{loader:'vue-loader'}

]

},

{

test:/\.js$/,

exclude: /node\_modules/,

use:[{loader:'babel-loader'}]

}

]

},

//配置可省略的后缀

resolve:{

extensions:['.js','.json','.css','.vue'],

alias:{

'vue$':'vue/dist/vue.esm.js'

}

},

//配置webpack插件

plugins:[

//热加载插件

new webpack.HotModuleReplacementPlugin(),

//自动注入

new HtmlWebpackPlugin({

filename:'index.html',

template:'index.html',

inject:true

})

]

}

参考文档：[webpack vue 热部署配置 - 简书 (jianshu.com)](https://www.jianshu.com/p/6f6588fddcea)

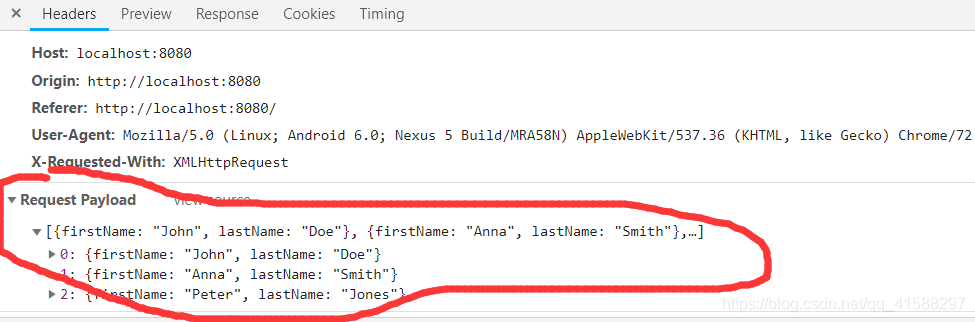
## 前端向后端传输数据的格式

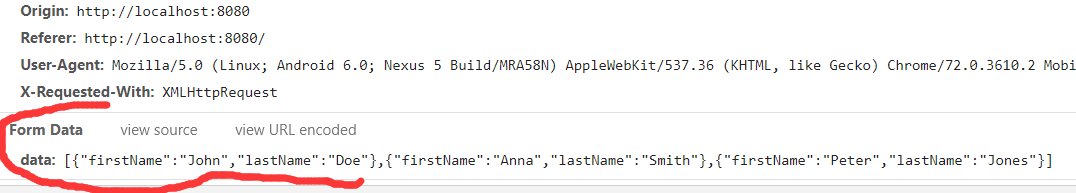
1、以查询字符串的方式传入

2、url传参的方式

3、以form\_data方式传输

4、以json数据传输 JSON格式数据是request payload



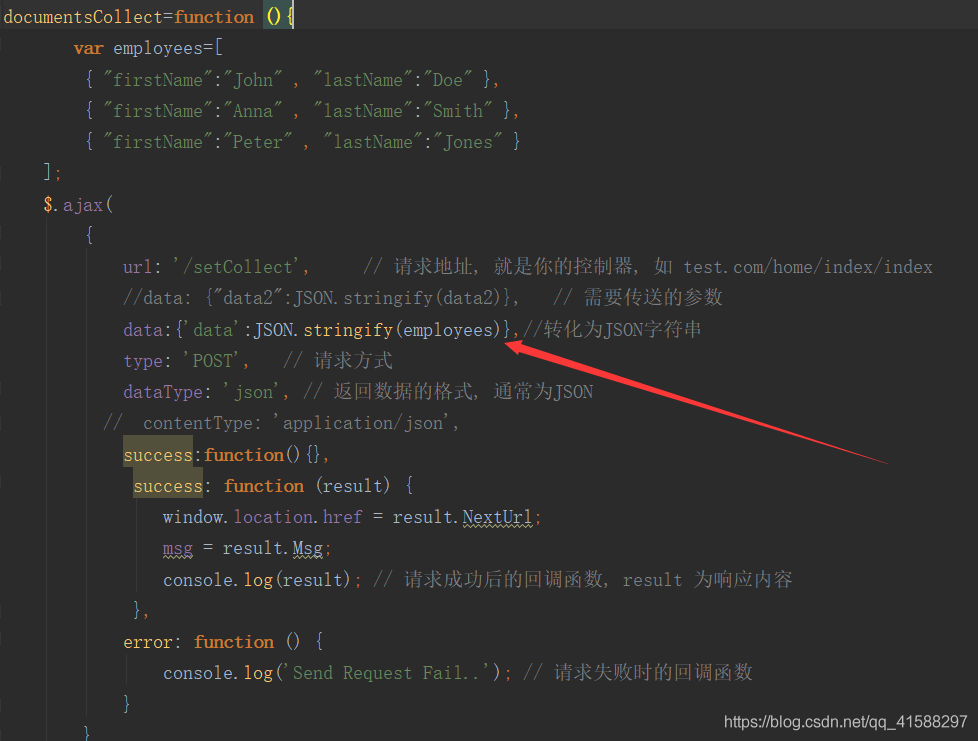


### 区别

二者区别在于：对于后台来说FormData中是键值对。在后台可以利用request.getParameter("data")来获取到后面的那些值,因为在Request中会自动转化Form Data为key:value。而Request payload却不可以，因为是JSON格式,Reuqest中不存在该键值对。

### 应用

在Ajax中向后台传数据的时候：可以自行的选择利用那种方式：如下：



#### 注意：

有json数组employees。我们需要将这个值传到后台。

1：利用一般方式formData：该方式需要将上面的contentType: 'application/json'注释掉，不注释掉的意思就是利用JSON(request payload)格式传输。然后data部分的值为data:{'data':JSON.stringify(employees)}。其中有引号的‘data’是自定义的键，也就是后台request.getParaml(”data")中的data，JSON.stringify(employees)就是值，是要传的数据，意思是将数组序列化为字符串。

详解：<https://blog.csdn.net/qq_41588297/article/details/86558731>

## Vue qs模块

**qs 是一个增加了一些安全性的查询字符串解析和序列化字符串的库。**

**qs.parse()是将URL解析成对象的形式**  
**qs.stringify()是将对象 序列化成URL的形式，以&进行拼接**

## **vue表单提交**





## ****watch+computed+methods****

### 总结：

1. 计算属性computed在使用时，一定要注意，函数里面的变量都会被监听，只要里面的某一个值变动，便会将整个函数执行一遍。 而 watch 只是监听某一个值，若是监听的值里面也有很多变量，也会全部监听  
  2. 计算后的属性可不在 data 中定义，如果定义会报错，因为对应的computed作为计算属性定义并返回对应的结果给这个变量,变量不可被重复定义和赋值。 而 watch 监听 data 中定义的变量变化  
 3、方法是每次调用都会执行函数

**计算属性和监听的区别**：计算属性不能执行异步任务，计算属性必须同步执行。也就是说计算属性不能向服务器请求或者执行异步任务。如果遇到异步任务，就交给侦听属性。Watch也可以检测computed属性。

### Computed 特性：

1.是计算值，  
2.应用：就是简化tempalte里面{{}}计算和处理props或$emit的传值  
3.具有缓存性，页面重新渲染值不变化,计算属性会立即返回之前的计算结果，而不必再次执行函数

接下来介绍各自的方法：

#### 计算属性computed

**在一个计算属性里可以完成各种复杂的逻辑，包括运算、函数调用等，只要最终返回一个结果就可以。**

<div id="example">

<p>Original message: "{{ message }}"</p>

<p>Computed reversed message: "{{ reversedMessage }}"</p> // 我们把复杂处理放在了计算属性里面了

</div>

**计算属性还可以依赖多个Vue 实例的数据**只要其中任一数据变化，计算属性就会重新执行，视图也会更新。

每一个计算属性都包含一个getter和一个setter ，我们上面的两个示例都是计算属性的默认用法， 只是利用了getter 来读取。

在你需要时，也可以提供一个setter 函数， 当手动修改计算属性的值就像修改一个普通数据那样时，就会触发setter 函数，执行一些自定义的操作，例如：

<div id="demo">

<p> {{ fullName }} </p>

<input type="text" v-model="fullName">

<input type="text" v-model="firstName">

<input type="text" v-model="lastName">

</div>

<script>

var vm = new Vue({

el: '#demo',

data: {

firstName: 'zhang',

lastName: 'san'

},

computed: {

fullName: {

//getter 方法

get(){

console.log('computed getter...')

return this.firstName + ' ' + this.lastName

},

//setter 方法

set(newValue){

console.log('computed setter...')

var names = newValue.split(' ')

this.firstName = names[0]

this.lastName = names[names.length - 1]

return this.firstName + ' ' + this.lastName

}

}

},

updated () {

console.log('updated')

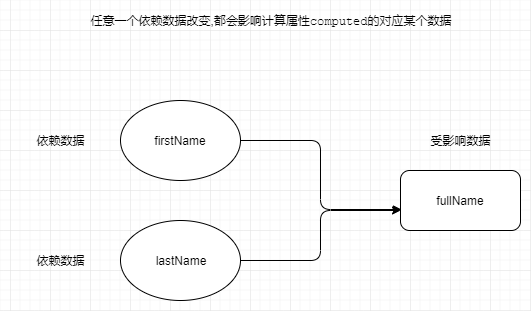
}

})

</script>

我们可以看到，input 是直接绑 v-model="fullName"，如果我们这里直接修改了fullName的值，那么就会触发setter，同时也会触发getter以及updated函数。其执行顺序是setter -> getter -> updated，如下：

注意：并不是触发了setter也就会触发getter，他们两个是相互独立的。我们这里修改了fullName会触发getter是因为setter函数里有改变firstName 和 lastName 值的代码。



### Methods

除了使用计算属性外，我们也可以通过在表达式中调用方法来达到同样的效果，如：

<div>{{reverseTitle()}}</div>

methods: {

reverseTitle: function () {

return this.title.split('').reverse().join('')

}

}

我们可以将同一函数定义为一个方法而不是一个计算属性，两种方式的最终结果确实是完全相同的。只是一个使用reverseTitle()取值，一个使用reverseTitle取值。

然而，不同的是计算属性是基于它们的依赖进行缓存的。计算属性只有在它的相关依赖发生改变时才会重新求值。

这就意味着只要 title还没有发生改变，多次访问 reverseTitle计算属性会立即返回之前的计算结果，而不必再次执行函数。

### Watch特性

监听对象里面某个属性



1.是观察的动作，  
2.应用：监听props，$emit或本组件的值执行异步操作  
3.无缓存性，页面重新渲染时值不变化也会执行

#### 监听属性watch

##### watch监控自身属性变化：

new Vue({

el: '#app',

data: {

firstName: 'hello',

lastName: 'vue',

fullName: 'hello.ve'

},

watch: {

'firstName': function(newval, oldval) {

// console.log(newval,oldval);

this.fullName = this.firstName + this.lastName;

},

'lastName': function(newval, oldval) {

// console.log(newval,oldval);

this.fullName = this.firstName + this.lastName;

}

}

});

##### watch监控路由对象：

new Vue({

el: '#app',

router: router, //开启路由对象

watch: {

'$route': function(newroute, oldroute) {

console.log(newroute, oldroute);

//可以在这个函数中获取到当前的路由规则字符串是什么

//那么就可以针对一些特定的页面做一些特定的处理

}

}

})

##### Watch监听对象的单个属性

watch如果想要监听对象的单个属性的变化,必须用computed作为中间件转化,因为computed可以取到对应的属性值。

data(){

return{

'first':{

second:0

}

}

},

computed:{

secondChange(){

return this.first.second

}

},

watch:{

secondChange(){

console.log('second属性值变化了')

}

},

### 简单实现computed和watch

#### 公共类

function defineReactive(data, key, val, fn) {

let subs = [] // 新增

Object.defineProperty(data, key, {

configurable: true,

enumerable: true,

get: function() {

// 新增

if (data.$target) {

subs.push(data.$target)

}

return val

},

set: function(newVal) {

if (newVal === val) return

fn && fn(newVal)

// 新增

if (subs.length) {

// 用 setTimeout 因为此时 this.data 还没更新

setTimeout(() => {

subs.forEach(sub => sub())

}, 0)

}

val = newVal

},

})

}

#### Computed实现

function computed(ctx, obj) {

let keys = Object.keys(obj)

let dataKeys = Object.keys(ctx.data)

dataKeys.forEach(dataKey => {

defineReactive(ctx.data, dataKey, ctx.data[dataKey])

})

let firstComputedObj = keys.reduce((prev, next) => {

ctx.data.$target = function() {

ctx.setData({ [next]: obj[next].call(ctx) })

}

prev[next] = obj[next].call(ctx)

ctx.data.$target = null

return prev

}, {})

ctx.setData(firstComputedObj)

}

#### Watch实现

function watch(ctx, obj) {

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

defineReactive(ctx.data, key, ctx.data[key], function(value) {

obj[key].call(ctx, value)

})

})

}

## 组件通信

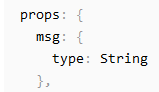
### 父组件->子组件

1. props

父组件



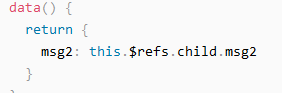
子组件



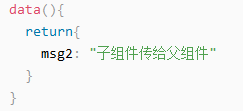
1. ref

父组件：





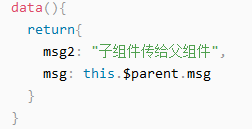
子组件：



1. childre、 parent

使用 parent 查找当前组件的父组件。  
使用 this.$children 查找当前组件的直接子组件，可以遍历全部子组件， 需要注意 $children 并不保证顺序，也不是响应式的，故不推荐使用。



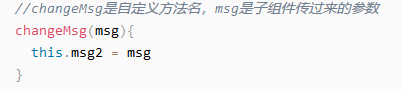


### 子组件-> 父组件

子组件



父组件

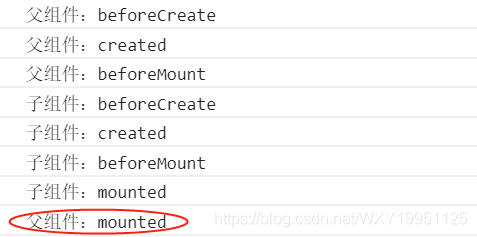


兄弟传参

1. 公共的bus.js
2. Vuex

### 父子组件调用方法的顺序

使用vue-cli创建的项目，父组件LeftMainmenu.vue里通过<router-view>加载了子组件，父组件与子组件的methods里各自有方法，初始化的时候在mounted里进行了调用。预设的想法是想在父组件LeftMainmenu.vue的方法里把axios请求到的数据设置成缓存【以下用localdata替代】，然后子组件调用的方法去使用localdata进行接下来的操作。但调试结果发现，无论父组件LeftMainmenu.vue里的方法放在函数生命周期里的哪个钩子里，执行的顺序总是慢于子组件的方法，导致子组件里调用的localdata数据为空……



### 解决思路

1、通过判断父组件挂载是否成功，成功后再去渲染子组件



2、在父组件mounted回调里执行先执行父组件的初始化函数，然后执行子组件的初始化函数（或者在父组件的axios成功回调里再执行子组件的初始化函数）

这种方法需要给放子组件的标签添加一个ref属性<router-view ref="son">，通过this.$refs.son.getsubData();【getsubData为子组件的方法名，也可以传参数】。父组件调用如下图【其中getJson是父组件请求接口获取localdata的方法】



3、所有非必要在子组件处理的数据都放在父组件里面里进行处理然后通过组件通信的方式传递给子组件

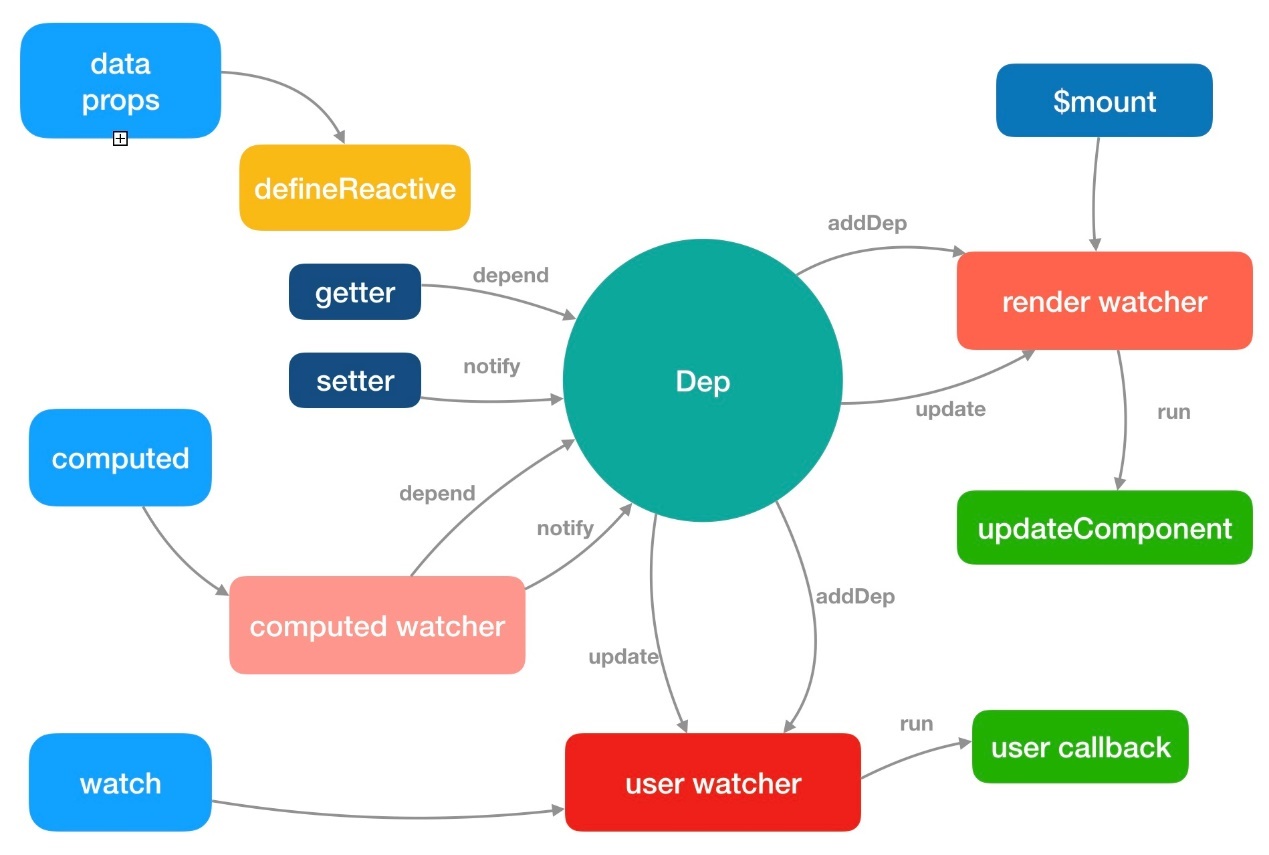
## 深入剖析vue



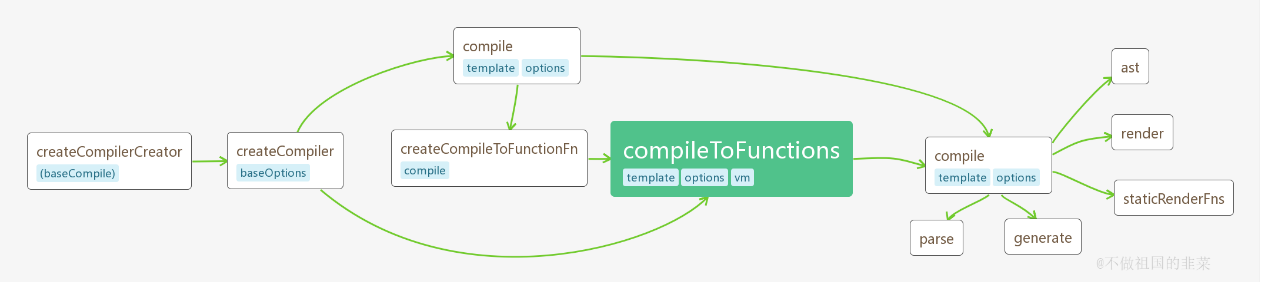
### Vue 本质

Vue本质是一个构造器，并且它保证了只能通过new实例的形式去调用，而不是直接通过函数的形式去使用。

#### Vue原理图

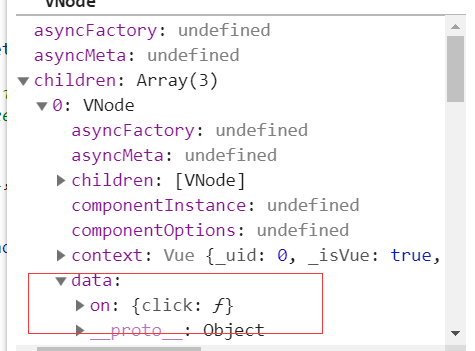


### Vue编译流程图



### 深入剖析 事件绑定

前面花了大量的篇幅介绍了模板上的事件标记在构建AST树上是怎么处理，并且如何根据构建的AST树返回正确的render渲染函数，**但是真正事件绑定还是离不开绑定注册事件**。这一个阶段就是发生在组件挂载的阶段。有了render函数，自然可以生成实例挂载需要的Vnode树，并且会进行patchVnode的环节进行真实节点的构建，如果发现过程已经遗忘，可以回顾以往章节。Vnode树的构建过程和之前介绍的内容没有明显的区别，所以这个过程就不做赘述，最终生成的vnode如下



### 深入剖析 v-for ( 时间循环 ) 与虚拟dom

#### Key

Expect ： number | string | Boolean(2.4.2 新增) | symbol (2.5.12 新增)

Key 的特殊 attribute 主要用在 Vue 虚拟dom 算法， 在新旧nodes 对比时辨识 VNodes。 如果不使用key， Vue会使用一种最大限度减少动态元素并且尽可能的尝试就地修改/ 复用相同类型元素的算法。而使用key 时 ，它会基于 key 的变化重新排列元素顺序， 并且会移除key 不存在的元素。

有相同父元素的子元素必须有独特的key。 重复的key 会造成渲染错误。

<ul>

<li v-for="item in items" :key="item.id">...</li>

</ul>

它也可以用于强制替换元素/ 数组而不是重复使用它。 当你遇到如下场景时它可能会很有用：

* 完整地触发组件的生命周期钩子
* 触发过渡

例如：

<transition>

<span :key="text">{{ text }}</span>

</transition>

当text 发生变化时， <span> 总会被替换而不是被修改，因此会触发过度

# Vue3.0

详解：<https://www.csdn.net/gather_2c/MtjaYgysOTEzNzAtYmxvZwO0O0OO0O0O.html>

完整的 API：[vue-composition-api-rfc.netlify.app/zh/api.html](https://vue-composition-api-rfc.netlify.app/zh/api.html" \t "https://www.csdn.net/gather_2c/_blank)

Vue3 template explorer ：

## 项目搭建

目前 vue3.0 的版本已经发布了  此次除了可以通过 vue-cli 和 webpack 搭建脚手架外 官方还提供了一种新的脚手架搭建工具 **vite**

### Vue 3.0 vite

#### Vite前景

Vite 是 Vue 作者开发的一款意图取代 webpack 的工具

其实现原理是利用 ES6 的 import 会发送请求去加载文件的特性，拦截这些请求，做一些预编译，省去 webpack 冗长的打包时间

vite 是一个基于浏览器原生 esmodule 的特性来工作的前端开发构建工具，并且通过 rollup来实现生产打包

本身 vite 是不需要将 **import/export** 等es语法转成 AMD 规范让浏览器去读取，而靠浏览器原生的引模块的能力去引入，这和 webpack 先在内存编译再发送给浏览器读取是有本质区别的，带来的好处是显而易见的，少了编译这一环节，浏览器的响应时间会大大缩短

vite 本身并不提供 **服务器能力** 和 **打包能力**，而是借助第三方包 koa 和 rollup 来实现，而真正核心的关键点，也是vite真正在做的是一整套中间件和插件系统

-----摘自知乎

#### 安装vite

npm install -g create-vite-app

create-vite-app projectName

cd projectName

npm install

npm run dev

#### vite工具相较于webpack的优缺点

##### 对比差异点

打开生成的项目过后，你会发现就是一个很普通的 Vue.js 应用，没有太多特殊的地方。

不过相比于之前 vue-cli 创建的项目或者是基于 Webpack 搭建的 Vue.js 项目，这里的开发依赖非常简单，只有 vite 和 @vue/compiler-sfc。

vite 就是我们今天要介绍的主角，而 @vue/compiler-sfc 就是用来编译我们项目中 .vue 结尾的单文件组件（SFC），它取代的就是 Vue.js 2.x 时使用的 vue-template-compiler。

再者就是 Vue.js 的版本是 3.0。这里尤其需要注意：**Vite 目前只支持 Vue.js 3.0 版本。**

#### **Vite编译模式**

即时编译（按需编译）

##### Optimize

Vite 还提供了一个目前在帮助列表中并没有呈现的一个子命令：optimize。

这个命令的作用就是单独的去「优化依赖」。

所谓的「优化依赖」，指的就是自动去把代码中依赖的第三方模块提前编译出来。

例如，我们在代码中通过 import 载入了 vue 这个模块，那通过这个命令就会自动将这个模块打包成一个单独的 ESM bundle, 放到 node\_modules/.vite\_opt\_cache 目录中。

这样后续请求这个文件时就不需要再即时去加载了。

##### HMR

同样也是模式的问题，热更新的时候，Vite 只需要立即编译当前所修改的文件即可，所以响应速度非常快。

而 Webpack 修改某个文件过后，会自动以这个文件为入口重写 build 一次，所有的涉及到的依赖也都会被加载一遍，所以反应速度会慢很多

##### Build

Vite 在生产模式下打包，需要使用 vite build 命令。

这个命令内部采用的是 Rollup 完成的应用打包，最终还是会把文件都提前编译并打包到一起。

对于 Code Splitting 需求，Vite 内部采用的就是原生 Dynamic imports 特性实现的，所以打包结果还是只能够支持现代浏览器。

不过好在 Dynamic imports 特性是可以有 Polyfill 的：[https://github.com/GoogleChromeLabs/dynamic-import-polyfill，也就是说，只要你想，它也可以运行在相对低版本的浏览器中。](https://github.com/GoogleChromeLabs/dynamic-import-polyfill%EF%BC%8C%E4%B9%9F%E5%B0%B1%E6%98%AF%E8%AF%B4%EF%BC%8C%E5%8F%AA%E8%A6%81%E4%BD%A0%E6%83%B3%EF%BC%8C%E5%AE%83%E4%B9%9F%E5%8F%AF%E4%BB%A5%E8%BF%90%E8%A1%8C%E5%9C%A8%E7%9B%B8%E5%AF%B9%E4%BD%8E%E7%89%88%E6%9C%AC%E7%9A%84%E6%B5%8F%E8%A7%88%E5%99%A8%E4%B8%AD%E3%80%82)

##### 打包or不打包的驳论

Vite 的出现，引发了另外一个值得我们思考的问题：究竟还有没有必要打包应用？

之前我们使用 Webpack 打包应用代码，使之成为一个 bundle.js，主要有两个原因：

1. 浏览器环境并不支持模块化
2. 零散的模块文件会产生大量的 HTTP 请求

随着浏览器的对 ES 标准支持的逐渐完善，第一个问题已经慢慢不存在了。现阶段绝大多数浏览器都是支持 ES Modules 的。

零散模块文件确实会产生大量的 HTTP 请求，而大量的 HTTP 请求在浏览器端就会并发请求资源的问题；

如上图所示，红色圈出来的请求就是并行请求，但是后面的请求就因为域名链接数已超过限制，而被挂起等待了一段时间。

在 HTTP 1.1 的标准下，每次请求都需要单独建立 TCP 链接，经过完整的通讯过程，非常耗时；

而且每次请求除了请求体中的内容，请求头中也会包含很多数据，大量请求的情况下也会浪费很多资源。

但是这些问题随着 HTTP 2 的出现，也就不复存在了。

关于 HTTP 1.1 与 HTTP 2 之间的差异，可以通过这个链接体验：[https://http2.akamai.com/demo，直观感受下](https://http2.akamai.com/demo%EF%BC%8C%E7%9B%B4%E8%A7%82%E6%84%9F%E5%8F%97%E4%B8%8B) HTTP/2 比 HTTP/1 到底快了多少。

而且不打包也有一个好处，就是可以把按需加载实现到极致

关于 HTTP 2 的详细介绍，可以参考：

* <https://blog.fundebug.com/2019/03/07/understand-http2-and-http3/>
* <https://www.digitalocean.com/community/tutorials/http-1-1-vs-http-2-what-s-the-difference>

##### 开箱即用

* TypeScript - 内置支持
* less/sass/stylus/postcss - 内置支持（需要单独安装所对应的编译器）

Vite 带来的优势主要体现在提升开发者在开发过程中的体验。

* Dev Server 无需等待，即时启动；
* 几乎实时的模块热更新；
* 所需文件按需编译，避免编译用不到的文件；
* 开箱即用，避免各种 Loader 和 Plugin 的配置；

#### Vite原理

Vite 的核心功能：Static Server + Compile + HMR

核心思路：

1. 将当前项目目录作为静态文件服务器的根目录
2. 拦截部分文件请求
   1. 处理代码中 import node\_modules 中的模块
   2. 处理 vue 单文件组件（SFC）的编译
3. 通过 WebSocket 实现 HMR

#### 手写实现

**#!/usr/bin/env node**

**const** path = require('path')

**const** { Readable } = require('stream')

**const** Koa = require('koa')

**const** send = require('koa-send')

**const** compilerSfc = require('@vue/compiler-sfc')

**const** cwd = process.cwd()

**const** streamToString = stream => **new** Promise((resolve, reject) => {

**const** chunks = []

stream.on('data', chunk => chunks.push(chunk))

stream.on('end', () => resolve(Buffer.concat(chunks).toString('utf8')))

stream.on('error', reject)

})

**const** app = **new** Koa()

*// 重写请求路径，/@modules/xxx => /node\_modules/*

app.use(**async** (ctx, next) => {

**if** (ctx.path.startsWith('/@modules/')) {

**const** moduleName = ctx.path.substr(10) *// => vue*

**const** modulePkg = require(path.join(cwd, 'node\_modules', moduleName, 'package.json'))

ctx.path = path.join('/node\_modules', moduleName, modulePkg.module)

}

**await** next()

})

*// 根据请求路径得到相应文件 /index.html*

app.use(**async** (ctx, next) => {

*// ctx.path // http://localhost:3080/*

*// ctx.body = 'my-vite'*

**await** send(ctx, ctx.path, { root: cwd, index: 'index.html' }) *// 有可能还需要额外处理相应结果*

**await** next()

})

*// .vue 文件请求的处理，即时编译*

app.use(**async** (ctx, next) => {

**if** (ctx.path.endsWith('.vue')) {

**const** contents = **await** streamToString(ctx.body)

**const** { descriptor } = compilerSfc.parse(contents)

**let** code

**if** (ctx.query.type === undefined) {

code = descriptor.script.content

code = code.replace(/export\s+default\s+/, 'const \_\_script = ')

code += `

import { render as \_\_render } from "${ctx.path}?type=template"

\_\_script.render = \_\_render

export default \_\_script`

*// console.log(code)*

ctx.type = 'application/javascript'

ctx.body = Readable.from(Buffer.from(code))

} **else** **if** (ctx.query.type === 'template') {

**const** templateRender = compilerSfc.compileTemplate({ source: descriptor.template.content })

code = templateRender.code

}

ctx.type = 'application/javascript'

ctx.body = Readable.from(Buffer.from(code))

}

**await** next()

})

*// 替换代码中特殊位置*

app.use(**async** (ctx, next) => {

**if** (ctx.type === 'application/javascript') {

**const** contents = **await** streamToString(ctx.body)

ctx.body = contents

.replace(/(from\s+['"])(?![\.\/])/g, '$1/@modules/')

.replace(/process\.env\.NODE\_ENV/g, '"production"')

}

})

app.listen(3080)

console.log('Server running @ http://localhost:3080')

#### vite文件处理机制

vite会先对文件类型做不同的处理

const resolvedPlugins = [

/\*\*

\* sourceMap 中间件和重写文件的中间件，为了等待其他中间件完成工作之后，返回给浏览器之前做处理

\*/

sourceMapPlugin,

moduleRewritePlugin,

htmlRewritePlugin,

// user plugins

...toArray(configureServer),

envPlugin,

...省略其他中间件

]

其中 moduleRewritePlugin 为重写 js 模块的 **核心中间件**，在所有中间件执行完毕之后，为所有 import 的模块打上标记和处理 **模块路径** 。vite 采用 es-module-lexer 来解析 **es** 模块**import** 信息。

静态资源文件的处理：静态资源通过路径的解析，直接返回资源绝对路径

//代码中写

import './assets/logo.png'

//实际请求 url

/src/assets/logo.png?import

//实际返回

export default "/src/assets/logo.png"

处理html 文件用的中间件为 rewriteHtml，逻辑很简单就做了三件事：

* 按 moduleRewritePlugin 的逻辑转换 script 标签里可能存在的 import 语句
* 为 hmr 热更新注册脚本，将script 的 src 属性 注册进 importerMap
* 为 hmr 热更新添加客户端脚本，这个将在热更新章节详细讲解

.vue 文件的解析，毕竟 vite 一开始就是为 vue3 保驾护航的，解析 .vue 文件主要用到了中间件 vuePlugin。当 .vue 文件到达服务器，vite首先会解析 SFC 的内容

然后通过 compileSFCMain 这个解析函数，将 .vue 文件中的 script标签，解析成对应的 js 模块，当初次解析 .vue 文件时，会对 style、template 进行 query 标记

客户端通过重写过的 .vue 文件，再去向服务端请求对应的 style、template 模块

import "/src/App.vue?type=style&index=0"

import { render as \_\_render } from "/src/App.vue?type=template"

由于有 query 标记，通过 compileSFCStyle解析 style 标签，compileSFCTemplate 解析 template标签

style 文件通过 cssPlugin 中间件来解析，通过 processCss 方法来解析 css ，通过 codegenCss 方法来重写 css 文件为 esmodule。

##### Vite的热更新机制

vite 的热更新可以实现代码效果的 **毫秒级** 响应，告别痛苦的等编译时间，实现真正意义上的 **热更新**，接下来让我一起来探究其 **热更新** 的实现原理~

###### 热更新概念

**热更新（hot module replacement）**，简称 hmr ，是一种无需刷新浏览器即可更新代码效果的技术，实现该技术的关键点是要建立 **浏览器** 和 **服务器** 之间的联系，还好我们现成就有一种技术可以实现：websocket协议，普通的 http 协议为短连接，一次会话结束就会关闭，这显然没法满足我们时刻都需要关联 **浏览器**和 **服务器** 的需求，而websocket 为长连接，可以一直保持 **浏览器** 和 **服务器** 之间的会话不中断，通过事件来互相传送数据，vite 也用了 websocket 来实现 **热更新**。

###### Websocket连接方式

之前说过 vite 通过 websocket 来实现 **浏览器** 和 **服务器** 之间的链接，具体到源码里，服务端在 hmrPlugin 中间件里

/\*\*=== hmrPlugin ===\*\*/

//...省略部分代码

// 创建 WebSocket 服务

const wss = new WebSocket.Server({ noServer: true })

// WebSocket 和客户端连接成功的事件

wss.on('connection', (socket) => {

debugHmr('ws client connected')

socket.send(JSON.stringify({ type: 'connected' }))

})

// WebSocket 异常事件

wss.on('error', (e: Error & { code: string }) => {

if (e.code !== 'EADDRINUSE') {

console.error(chalk.red(`[vite] WebSocket server error:`))

console.error(e)

}

})

客户端在client.ts文件中

/\*\*=== client.ts ===\*\*/

//...省略部分代码

const socketProtocol =

\_\_HMR\_PROTOCOL\_\_ || (location.protocol === 'https:' ? 'wss' : 'ws')

const socketHost = `${\_\_HMR\_HOSTNAME\_\_ || location.hostname}:${\_\_HMR\_PORT\_\_}`

// 创建 WebSocket 服务

const socket = new WebSocket(`${socketProtocol}://${socketHost}`, 'vite-hmr')

// 监听服务端发送的消息请求

socket.addEventListener('message', async ({ data }) => {

const payload = JSON.parse(data) as HMRPayload | MultiUpdatePayload

if (payload.type === 'multi') {

//如果有多个请求，就循环处理 payload.updates 更新队列

payload.updates.forEach(handleMessage)

} else {

//单个请求就直接处理

handleMessage(payload)

}

})

//监听是否关闭了连接

socket.addEventListener('close', () => {

console.log(`[vite] server connection lost. polling for restart...`)

setInterval(() => {

fetch('/')

.then(() => {

//关闭之后一秒，重新刷新页面，再次进行重连接

location.reload()

})

.catch((e) => {

/\* ignore \*/

})

}, 1000)

})

##### 监听文件变化

既然要 **热更新** ，光有 websocket 是不够的，因为我们需要一个 action ，也就是触发时机去触发事件，才能达到更新的目的。vite 选用了和 webpack 一样的解决方案 [chokidar](https://link.zhihu.com/?target=https%3A//github.com/paulmillr/chokidar" \t "_blank)，为啥用这个呢？我们知道 nodejs 本身有 fs 模块的 api 来实现文件监听的功能，但是因为缺少一系列的优化，会带来一系列的问题，chokidar 的出现就为了解决这个问题，保证原来功能的基础上做了一系列的优化，可以更快的响应，兼容性也刚好。

通过chokidar 初始化一个监听对象

/\*\*=== server/index.ts ===\*\*/

//...省略部分代码

const watcher = chokidar.watch(root, {

ignored: [/node\_modules/, /\.git/],

// #610

awaitWriteFinish: {

stabilityThreshold: 100,

pollInterval: 10

}

}) as HMRWatcher

const context: ServerPluginContext = {

root,

app,

server,

watcher,

resolver,

config,

port: config.port || 3000

}

app.use((ctx, next) => {

//将 ServerPluginContext 注入到 koa 上下文对象中

Object.assign(ctx, context)

ctx.read = cachedRead.bind(null, ctx)

return next()

})

然后在处理不同的中间件里，注入 watcher 的监控代码

/\*\*=== rewriteHtml ===\*\*/

watcher.on('change', (file) => {

const path = resolver.fileToRequest(file)

if (path.endsWith('.html')) {

debug(`${path}: cache busted`)

// html 文件 直接重新加载页面

watcher.send({

type: 'full-reload',

path

})

console.log(chalk.green(`[vite] `) + ` ${path} page reloaded.`)

}

})

/\*\*=== vuePlugin ===\*\*/

watcher.on('change', (file) => {

if (file.endsWith('.vue')) {

handleVueReload(file)

}

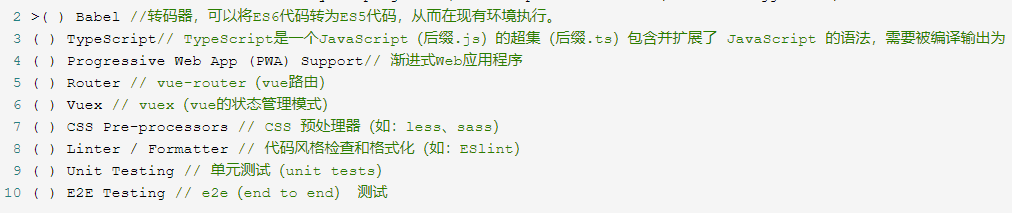
})

//etc

### Vue vue-cli

Vue3.0安装配置（使用vue-cli创建项目）

vue create



基础范例：



### Webpack创建项目

[vue-cli脚手架工具搭建vue-webpack项目\_个人文章 - SegmentFault 思否](https://segmentfault.com/a/1190000019701198)

1. 安装nodejs
2. 全局安装webpack

Npm install webpack -g

Npm install -g webpack

(可以使用cnpm代替npm)

1. 全局安装vue-cli

在cmd中输入命令：npm install –global vue-cli

1. 创建项目 : vue init webpack [项目名称]

Vue3.0 属性全局配置

 Vue 2.x  通过 Vue.prototype扩展

Vue3.0 通过app.config.globalProperties

## Setup钩子

setup函数是一个新的组件选项。作为在组件内使用 Composition API 的入口点

### 调用时机

创建组件实例，然后初始化 props ，紧接着就调用setup 函数。从生命周期钩子的视角来看，它会在 beforeCreate 钩子之前被调用

### 模板中使用

如果 setup 返回一个对象，则对象的属性将会被合并到组件模板的渲染上下文

<template>

<div>{{ count }} {{ object.foo }}</div>

</template>

<script>

import { ref, reactive } from 'vue'

export default {

setup() {

const count = ref(0)

const object = reactive({ foo: 'bar' })

// 暴露给模板

return {

count,

object,

}

},

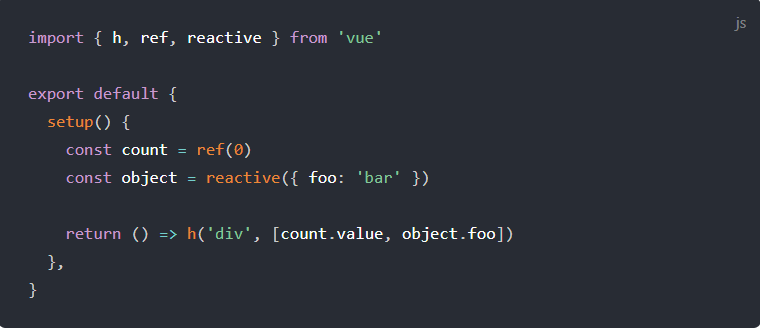
}

</script>

注意 setup 返回的 ref 在模板中会自动解开，不需要写 .value。

### 渲染函数/jsx中使用

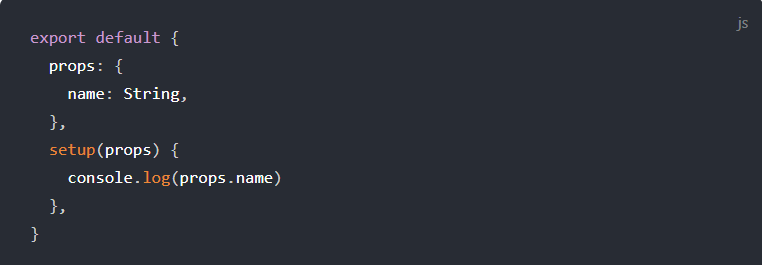
setup 也可以返回一个函数，函数中也能使用当前 setup 函数作用域中的响应式数据：



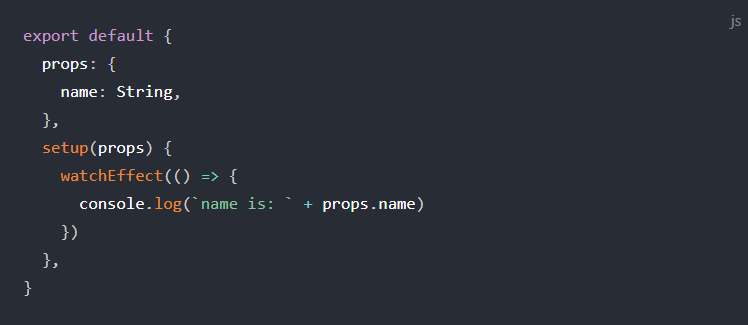
### 参数

Setup参数主要是两个接收参数（props）、上下文对象（context）

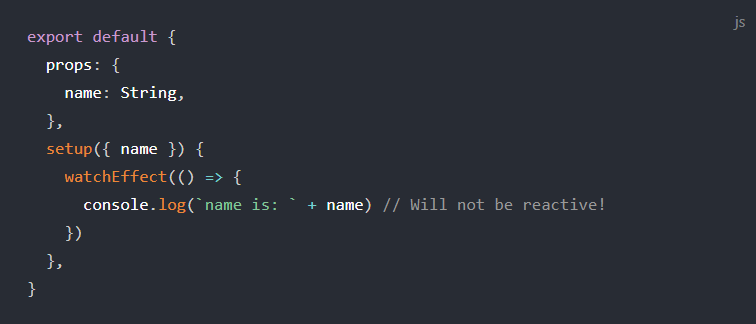
#### 接收参数



注意 props 对象是响应式的，watchEffect 或 watch 会观察和响应 props 的更新：



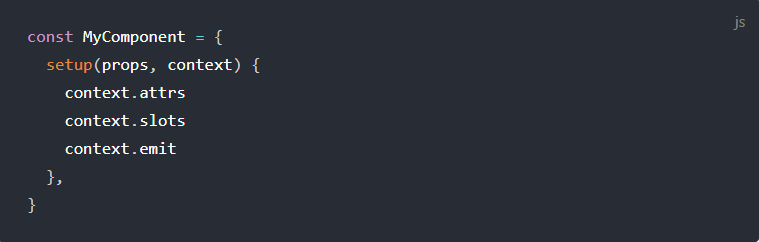
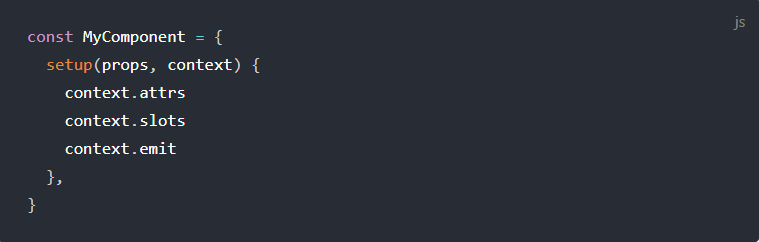
然而**不要**解构 props 对象，那样会使其失去响应性：



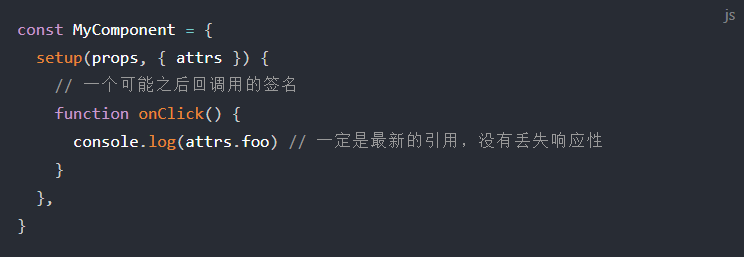
在开发过程中，props 对象对用户空间代码是不可变的（用户代码尝试修改 props 时会触发警告）。

### 上下文对象

第二个参数提供了一个上下文对象，从原来 2.x 中 this 选择性地暴露了一些 property。



attrs 和 slots 都是内部组件实例上对应项的代理，可以确保在更新后仍然是最新值。所以可以解构，无需担心后面访问到过期的值：

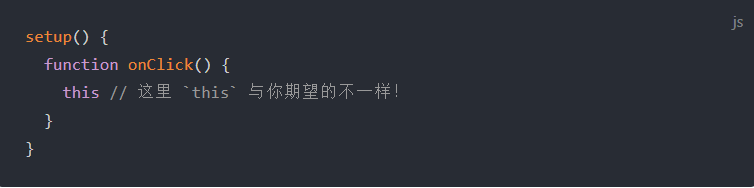


出于一些原因将 props 作为第一个参数，而不是包含在上下文中：

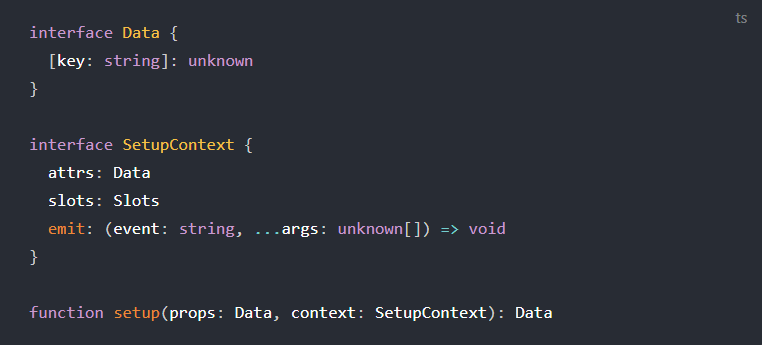
* 组件使用 props 的场景更多，有时候甚至只使用 props
* 将 props 独立出来作为第一个参数，可以让 TypeScript 对 props 单独做类型推导，不会和上下文中的其他属性相混淆。这也使得 setup 、 render 和其他使用了 TSX 的函数式组件的签名保持一致

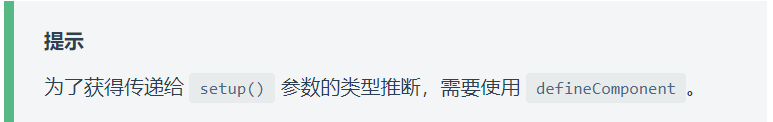
#### This的用法

**this 在 setup() 中不可用**。由于 setup() 在解析 2.x 选项前被调用，setup() 中的 this 将与 2.x 选项中的 this 完全不同。同时在 setup() 和 2.x 选项中使用 this 时将造成混乱。在 setup() 中避免这种情况的另一个原因是：这对于初学者来说，混淆这两种情况的 this 是非常常见的错误：



#### 类型定义





### watchEffect

<https://blog.csdn.net/weixin_43294560/article/details/107581527>

立即执行传入的一个函数，并响应式追踪其依赖，并在其依赖变更时重新运行该函数。

用法:

import { watchEffect } from 'vue'

setup()

{

watchEffect(函数(){

使用到了哪个ref/reactive对象.value,就监听哪个

})

(1)停止侦听

隐式停止:

当watchEffect在组件的setup()函数或生命周期钩子被调用时,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。

显示停止:

const xx=watchEffect(...)

xx.stop();

(2)清除副作用

watchEffect传入参数

watchEffect((onInvalidate)=>{

onInvalidate(()=>{

执行时机:

在副作用即将重新执行时

如果在setup()或生命周期钩子函数中使用了 watchEffect, 则在卸载组件时

})

})

(3)异步副作用

watchEffect(async () => {}

(4)副作用需要同步或在组件更新之前

watchEffect(

() => {

...

},

{

flush: 'sync', 'pre'组件更新前运行

}

)

(5)调试侦听器的行为

watchEffect(

() => {

...

},

{

onTrigger(e) {

依赖项变更导致副作用被触发时

e.target包含了值

},

onTrack(e){

当一个 reactive对象属性或一个 ref 作为依赖被追踪时触发

e.target包含了值

}

}

)

#### 代码范例



<template>

<div>

<img src="./logo.png">

<h1>Hello Vue 3!</h1>

{{name}}{{obj.sex}}

<button @click="inc">Clicked {{ count }} times.</button>

</div>

</template>

<script>

import { ref,reactive,computed,readonly,watchEffect } from 'vue'

export default {

setup() {

let count = ref(0)

let res=1;

let name = ref('jeff')

const obj=reactive({sex:'male'})

const robj=readonly(obj);

let r=readonly('aa') //不具有只读的能力

watchEffect((onInvalidate)=>{

console.log(count.value);

onInvalidate(()=>{

console.log('清除');

clearInterval(timer);

})

}, {

onTrigger(e) {

console.log(e);

},

onTrack(e)

{

console.log('triger');

console.log(e)

}

})

const inc = () => {

timer=setInterval(()=>{

count.value++;

},1000)

}

return {

count,

inc,

name, //在setup返回对象中自动解套

obj

}

}

}

</script>

<style scoped>

img {

width: 200px;

}

h1 {

font-family: Arial, Helvetica, sans-serif;

}

</style>

## 计算状态

## 响应式API

### Reactive

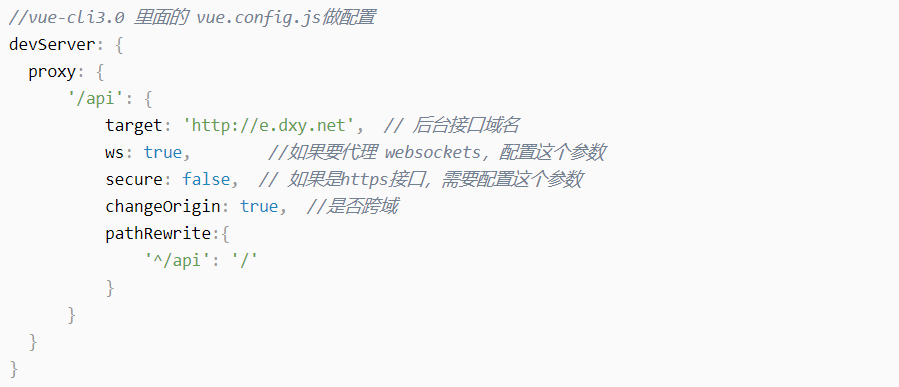
理解reactive： <https://www.cnblogs.com/binglove/p/13300793.html>

接收一个普通对象然后返回该普通对象的响应式代理。等同于 2.x 的 Vue.observable()

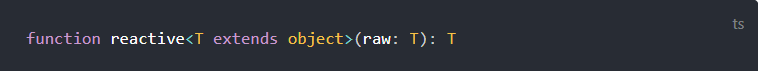


响应式转换是“深层的”：会影响对象内部所有嵌套的属性。基于 ES2015 的 Proxy 实现，返回的代理对象**不等于**原始对象。建议仅使用代理对象而避免依赖原始对象。

Vue代理配置

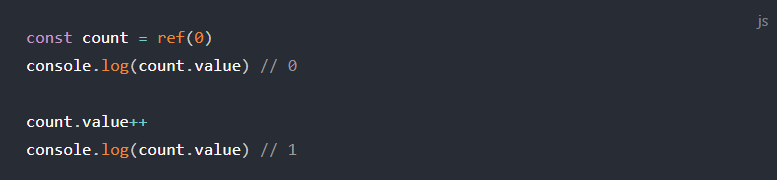


#### 类型定义



### Ref

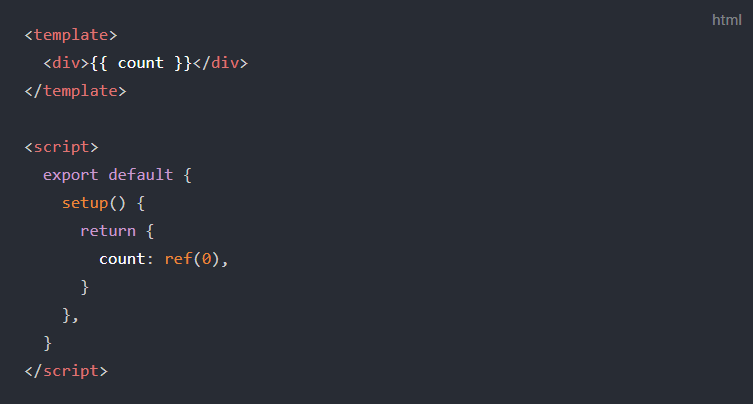
接受一个参数值并返回一个响应式且可改变的 ref 对象。ref 对象拥有一个指向内部值的单一属性 .value。



如果传入 ref 的是一个对象，将调用 reactive 方法进行深层响应转换。

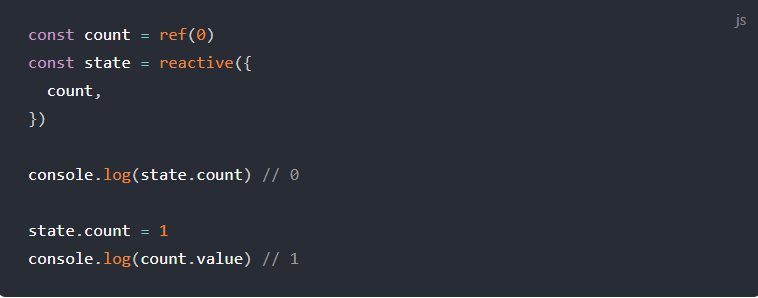
#### 模板中访问

当 ref 作为渲染上下文的属性返回（即在setup() 返回的对象中）并在模板中使用时，它会自动解套，无需在模板内额外书写 .value：

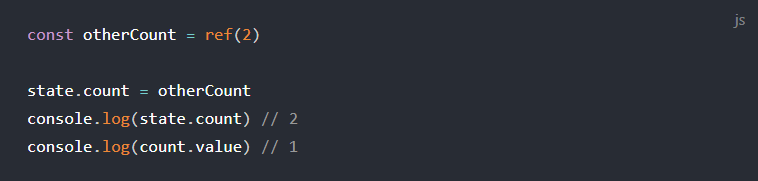


#### 作为响应式对象的属性访问

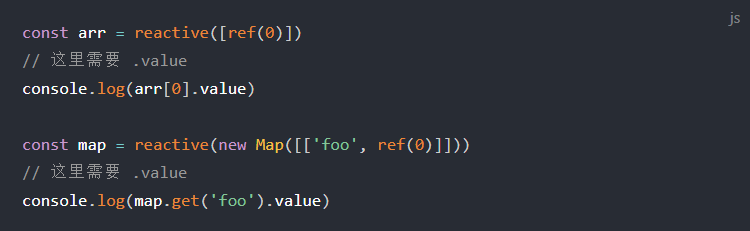
当 ref 作为 reactive 对象的 property 被访问或修改时，也将自动解套 value 值，其行为类似普通属性：



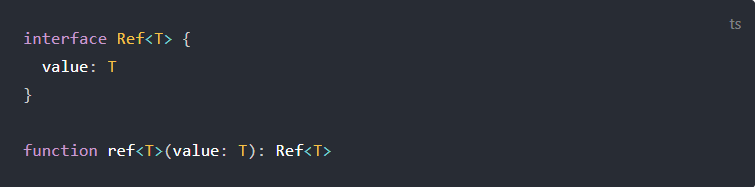
注意如果将一个新的 ref 分配给现有的 ref， 将替换旧的 ref：



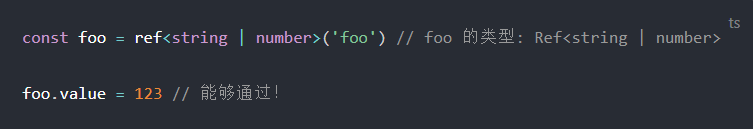
注意当嵌套在 reactive Object 中时，ref 才会解套。从 Array 或者 Map 等原生集合类中访问 ref 时，不会自动解套：



#### 类型定义

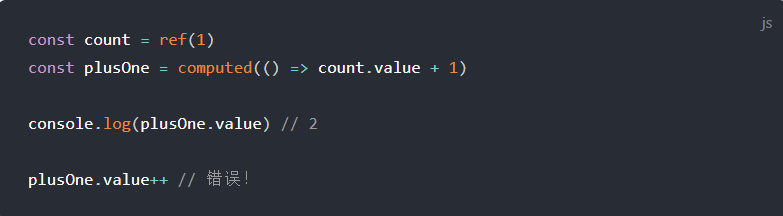


有时我们可能需要为 ref 做一个较为复杂的类型标注。我们可以通过在调用 ref 时传递泛型参数来覆盖默认推导：

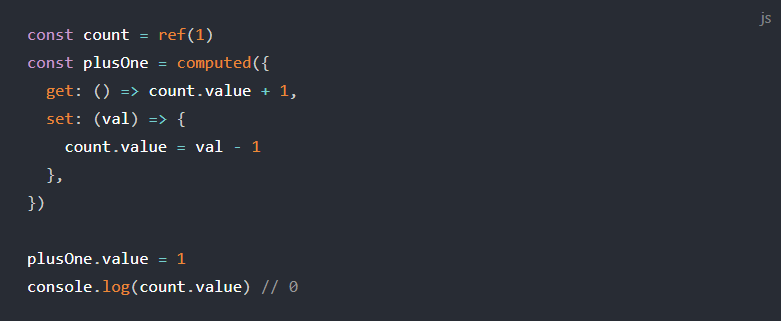


### Computed

传入一个 getter 函数，返回一个默认不可手动修改的 ref 对象。



或者传入一个拥有 get 和 set 函数的对象，创建一个可手动修改的计算状态。



#### 类型定义

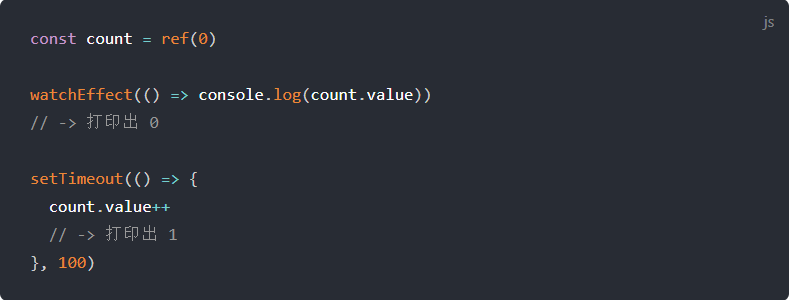
### Readonly

传入一个对象（响应式或普通）或 ref，返回一个原始对象的**只读**代理。一个只读的代理是“深层的”，对象内部任何嵌套的属性也都是只读的。



### watchEffect

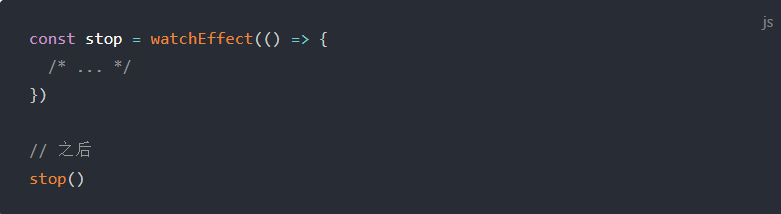
立即执行传入的一个函数，并响应式追踪其依赖，并在其依赖变更时重新运行该函数。



#### 停止侦听

当 watchEffect 在组件的 setup() 函数或生命周期钩子被调用时， 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：

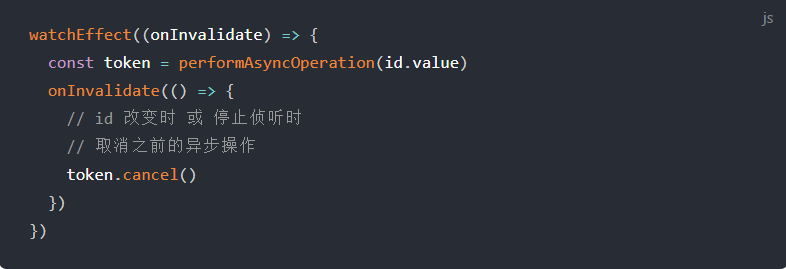


当停止监听手动执行后停止监听位置以下的代码不会再对watcheffect造成影响

#### 清除副作用

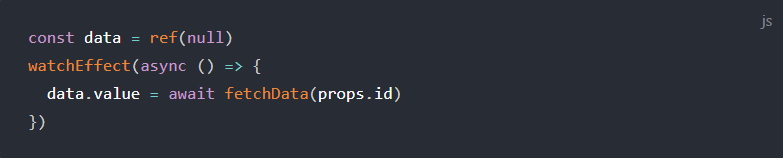
有时副作用函数会执行一些异步的副作用, 这些响应需要在其失效时清除（即完成之前状态已改变了）。所以侦听副作用传入的函数可以接收一个 onInvalidate 函数作入参, 用来注册清理失效时的回调。当以下情况发生时，这个**失效回调**会被触发:

* 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, 则在卸载组件时)



我们之所以是通过传入一个函数去注册失效回调，而不是从回调返回它（如 React useEffect 中的方式），是因为返回值对于异步错误处理很重要。

在执行数据请求时，副作用函数往往是一个异步函数：



我们知道异步函数都会隐式地返回一个 Promise，但是清理函数必须要在 Promise 被 resolve 之前被注册。另外，Vue 依赖这个返回的 Promise 来自动处理 Promise 链上的潜在错误。

#### 副作用刷新时机

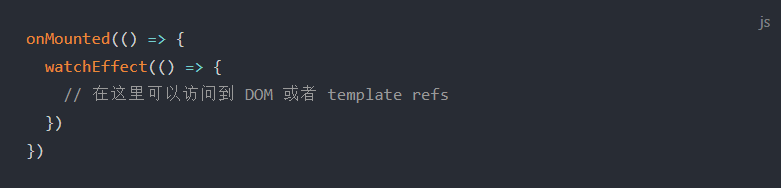
Vue 的响应式系统会缓存副作用函数，并异步地刷新它们，这样可以避免同一个 tick 中多个状态改变导致的不必要的重复调用。在核心的具体实现中, 组件的更新函数也是一个被侦听的副作用。当一个用户定义的副作用函数进入队列时, 会在所有的组件更新后执行：



在这个例子中：

* count 会在初始运行时同步打印出来
* 更改 count 时，将在组件**更新后**执行副作用。

请注意，初始化运行是在组件 mounted 之前执行的。因此，如果你希望在编写副作用函数时访问 DOM（或模板 ref），请在 onMounted 钩子中进行：



如果副作用需要同步或在组件更新之前重新运行，我们可以传递一个拥有 flush 属性的对象作为选项（默认为 'post'）：(flush 属性用来设置副作用执行时机)

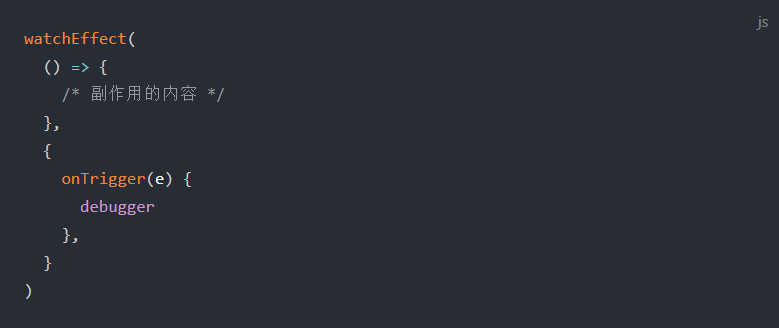


#### 侦听器调试

onTrack 和 onTrigger 选项可用于调试一个侦听器的行为。

* 当一个 reactive 对象属性或一个 ref 作为依赖被追踪时，将调用 onTrack
* 依赖项变更导致副作用被触发时，将调用 onTrigger

这两个回调都将接收到一个包含有关所依赖项信息的调试器事件。建议在以下回调中编写 debugger 语句来检查依赖关系：



**onTrack 和 onTrigger 仅在开发模式下生效。**

#### 类型定义



### Watch

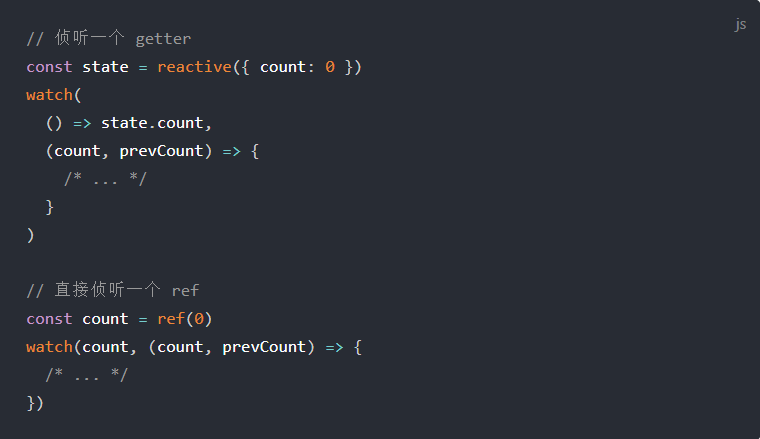
watch API 完全等效于 2.x this.$watch （以及 watch 中相应的选项）。watch 需要侦听特定的数据源，并在回调函数中执行副作用。默认情况是懒执行的，也就是说仅在侦听的源变更时才执行回调。

对比 watchEffect，watch 允许我们：

* 懒执行副作用；
* 更明确哪些状态的改变会触发侦听器重新运行副作用；
* 访问侦听状态变化前后的值。

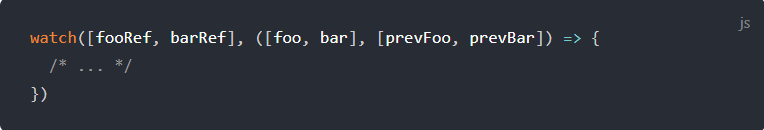
#### 侦听单个数据源

侦听器的数据源可以是一个拥有返回值的 getter 函数，也可以是 ref：



#### 侦听多个数据源

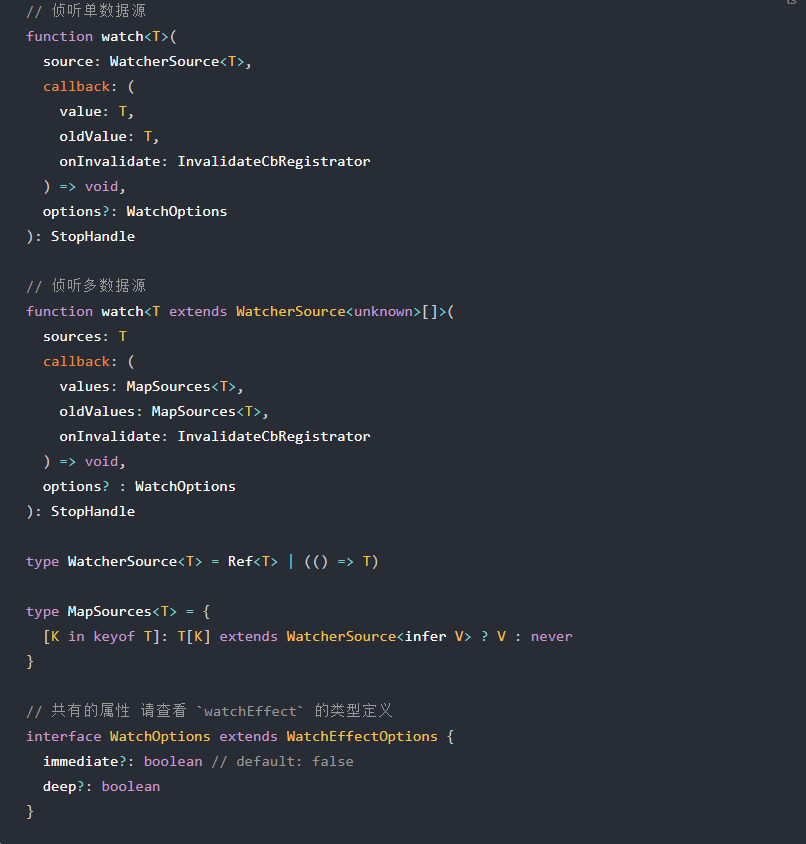
watcher 也可以使用数组来同时侦听多个源：



#### 与watchEffect共享的行为

watch 和 watchEffect 在[停止侦听](https://vue-composition-api-rfc.netlify.app/zh/api.html" \l "%E6%89%8B%E5%8A%A8%E5%81%9C%E6%AD%A2%E4%BE%A6%E5%90%AC), [清除副作用](https://vue-composition-api-rfc.netlify.app/zh/api.html#%E6%B8%85%E9%99%A4%E5%89%AF%E4%BD%9C%E7%94%A8) (相应地 onInvalidate 会作为回调的第三个参数传入)，[副作用刷新时机](https://vue-composition-api-rfc.netlify.app/zh/api.html" \l "%E5%89%AF%E4%BD%9C%E7%94%A8%E5%88%B7%E6%96%B0%E6%97%B6%E6%9C%BA) 和 [侦听器调试](https://vue-composition-api-rfc.netlify.app/zh/api.html#%E4%BE%A6%E5%90%AC%E5%99%A8%E8%B0%83%E8%AF%95) 等方面行为一致.

#### 类型定义



#### 生命周期钩子函数

可以直接导入 onXXX 一族的函数来注册生命周期钩子：



这些生命周期钩子注册函数只能在 setup() 期间同步使用， 因为它们依赖于内部的全局状态来定位当前组件实例（正在调用 setup() 的组件实例）, 不在当前组件下调用这些函数会抛出一个错误。

组件实例上下文也是在生命周期钩子同步执行期间设置的，因此，在卸载组件时，在生命周期钩子内部同步创建的侦听器和计算状态也将自动删除。

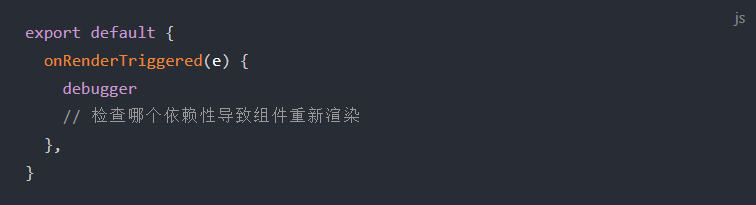
**与 2.x 版本生命周期相对应的组合式 API**

* ~~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**

除了和 2.x 生命周期等效项之外，组合式 API 还提供了以下调试钩子函数：

* onRenderTracked
* onRenderTriggered

两个钩子函数都接收一个 DebuggerEvent，与 watchEffect 参数选项中的 onTrack 和 onTrigger 类似：



## 依赖注入

provide 和 inject 提供依赖注入，功能类似 2.x 的 provide/inject。两者都只能在当前活动组件实例的 setup() 中调用。

依赖注入需要注意点：

provide：Object | () => Object

inject：Array<string> | { [key: string]: string | Symbol | Object }

provide：提供依赖是一个对象，或者是一个返回对象的函数。里面呢就包含要给子孙后代的东西，也就是属性和属性值。

inject： 注入依赖一个字符串数组，或者是一个对象。属性值可以是一个对象，包含from和default默认值。

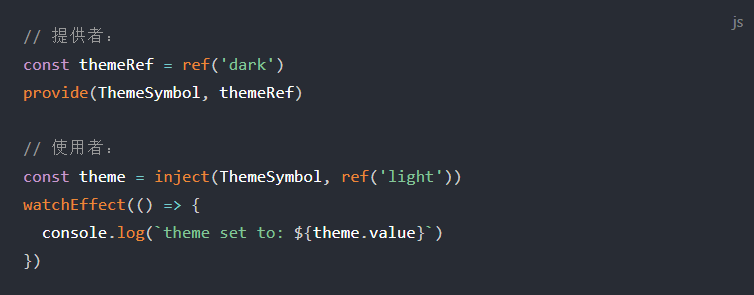


可以使用 ref 来保证 provided 和 injected 之间值的响应：

inject 接受一个可选的的默认值作为第二个参数。如果未提供默认值，并且在 provide 上下文中未找到该属性，则 inject 返回 undefined。

### 注入的响应性

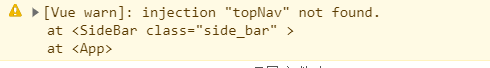
可以使用 ref 来保证 provided 和 injected 之间值的响应：（注入和接收之间的响应性：当接收方改变接受到的数据注入方注入数据也会发生改变并相应在界面上）



如果注入一个响应式对象，则它的状态变化也可以被侦听。

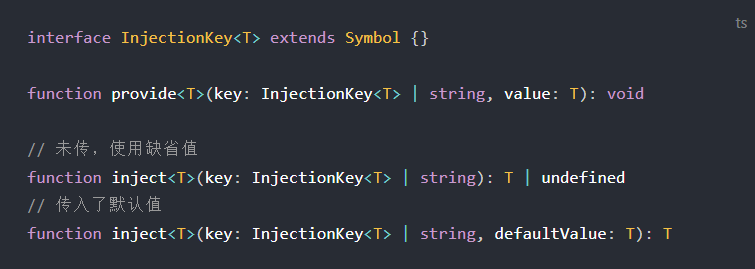
***思考：***如果注入数据为响应对象，是不是可也是用provided+injected替代vuex

**结论：**不可以，注入是从父级组件想子级组件进行的兄弟间是不可以进行的

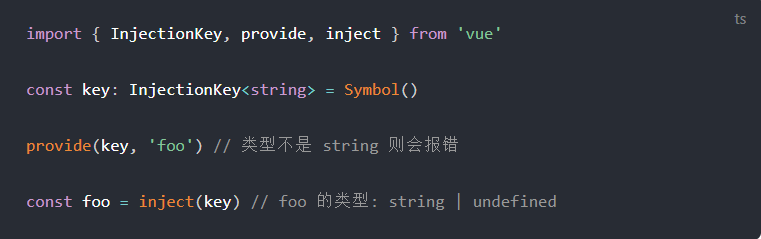


当兄弟间通过provide-inject进行通信的时候会报警告⚠如上图所示

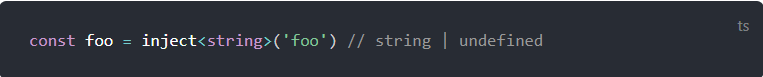
### 类型定义



Vue 提供了一个继承 Symbol 的 InjectionKey 接口。它可用于在提供者和消费者之间同步注入值的类型：



如果使用字符串作为键或没有定义类型的符号，则需要显式声明注入值的类型：



## Slot插槽

### 插槽含义：就是引入子组件后，在插入子组件元素中添加信息或者标签，使得子组件的指定位置插入信息或者标签

### 插槽有三种：默认插槽、具名插槽、作用域插槽，由于vue2.6.0后对插槽进行修改，但是兼容2.6.0前的版本，博文中只说明2.6.0后的插槽，vue3.0后面会去除2.60前的版本兼容

### 插槽分类

* 1. 匿名插槽
  2. 具名插槽
  3. 作用域插槽

**注意：**

vue2.6 使用 父组件slot=“name” slot-scope=“data” 可以使用div或template  
vue3.0 使用父组件v-slot:name="data"或者#name=“data”



### 重点：具名插槽+作用域插槽

## 模板refs

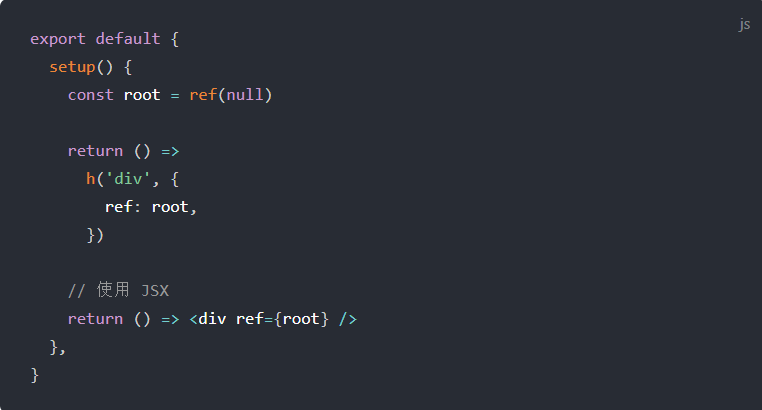
当使用组合式 API 时，*reactive refs* 和 *template refs* 的概念已经是统一的。为了获得对模板内元素或组件实例的引用，我们可以像往常一样在 setup() 中声明一个 ref 并返回它：



这里我们将 root 暴露在渲染上下文中，并通过 ref="root" 绑定到 div 作为其 ref。 在 Virtual DOM patch 算法中，如果一个 VNode 的 ref 对应一个渲染上下文中的 ref，则该 VNode 对应的元素或组件实例将被分配给该 ref。 这是在 Virtual DOM 的 mount / patch 过程中执行的，因此模板 ref 仅在渲染初始化后才能访问。

ref 被用在模板中时和其他 ref 一样：都是响应式的，并可以传递进组合函数（或从其中返回）。

### 配合render函数/JSX的用法



### 在v-for中使用

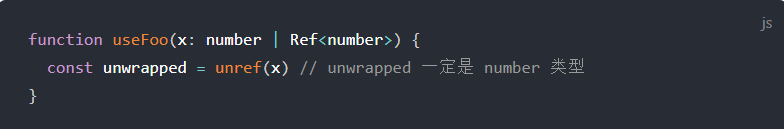
模板 ref 在 v-for 中使用 vue 没有做特殊处理，需要使用**函数型的 ref**（3.0 提供的新功能）来自定义处理方式：



## 响应式系统工具集

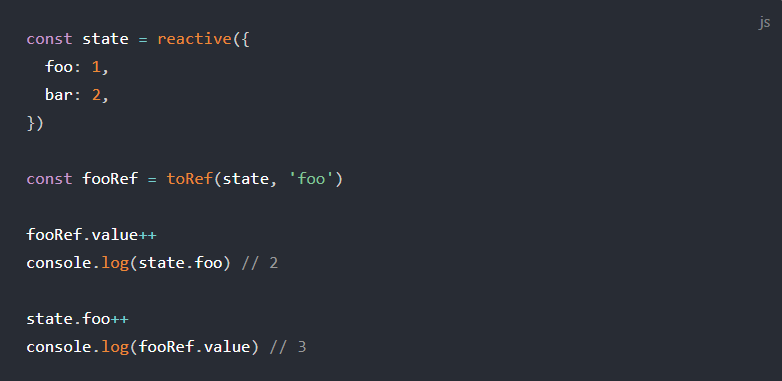
### Unref

如果参数是一个 ref 则返回它的 value，否则返回参数本身。它是 val = isRef(val) ? val.value : val 的语法糖。

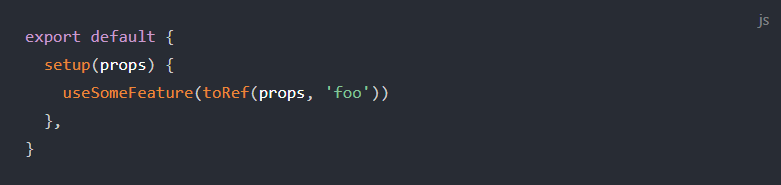


### toRef

toRef 可以用来为一个 reactive 对象的属性创建一个 ref。这个 ref 可以被传递并且能够保持响应性。



当您要将一个 prop 中的属性作为 ref 传给组合逻辑函数时，toRef 就派上了用场：



### 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 ，和响应式对象 property 一一对应

（虽然响应式对象转换为普通对象了但是，它的每一个property都是响应式（ref）的并且和转换前的响应式对象的property的一一对应）



当想要从一个组合逻辑函数中返回响应式对象时，用 toRefs 是很有效的，该 API 让消费组件可以 解构 / 扩展（使用 ... 操作符）返回的对象，并不会丢失响应性：



### isRef

检查一个值是否为一个ref对象

### isProxy

检查一个对象是否由reactive 或者readonly 方法创建的对象的代理

### isReactive

检查一个对象是否是由 reactive 创建的响应式代理。

如果这个代理是由 readonly 创建的，但是又被 reactive 创建的另一个代理包裹了一层，那么同样也会返回 true。

### isReadonly

检查一个对象是否由readonly创建的只读代理

## 高级响应式系统API

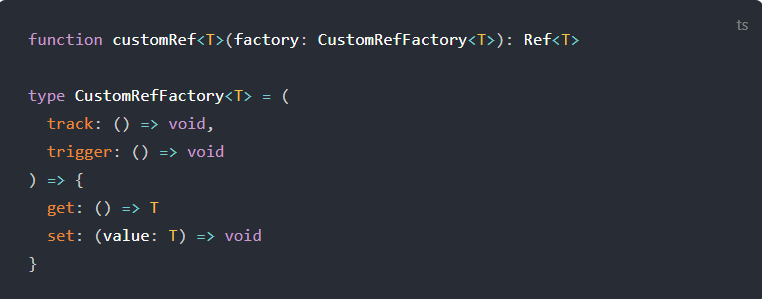
### customRef

customRef 用于自定义一个 ref，可以显式地控制依赖追踪和触发响应，接受一个工厂函数，两个参数分别是用于追踪的 track 与用于触发响应的 trigger，并返回一个带有 get 和 set 属性的对象。

#### 使用自定义 ref 实现带防抖功能的 v-model ：

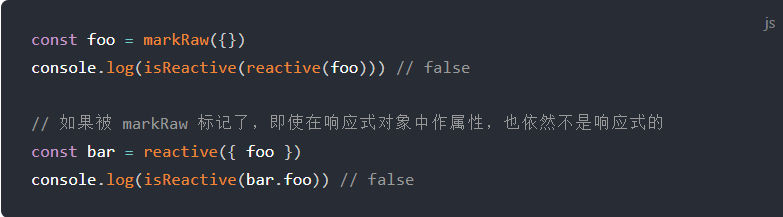


#### 类型定义



### markRaw

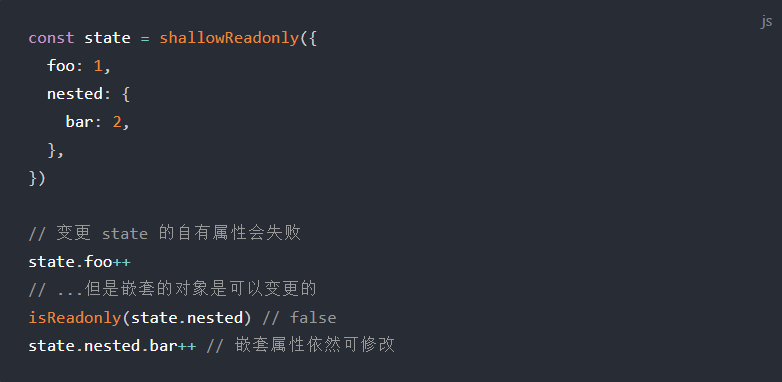
显式标记一个对象为“永远不会转为响应式代理”，函数返回这个对象本身。





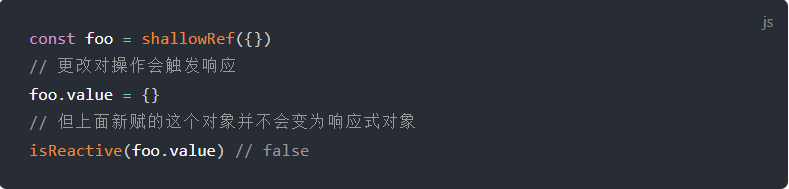
### shallowReactive

只为某个对象的自有（第一层）属性创建浅层的**只读**响应式代理，同样也不会做深层次、递归地代理，深层次的属性并不是只读的。



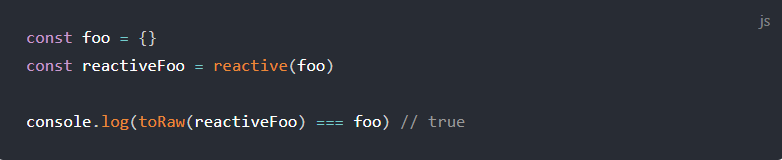
### shallowRef

创建一个 ref ，将会追踪它的 .value 更改操作，但是并不会对变更后的 .value 做响应式代理转换（即变更不会调用 reactive）



### 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，写入时也不会触发更改。不建议一直持有原始对象的引用。请谨慎使用。



## 相比于vue2.X vue3.0的优势

### Vue3.0六大亮点

 Performance: 性能比Vue 2.x快1.2~2倍

 Tree shaking support: 按需编译，体积比vue2.x更小

 Composition API: 组合API（类似React Hooks）

 Better TypeScript support: 更好的Ts支持

 Custom Renderer API: 暴露了自定义渲染API

 Fragment, Teleport(Protal), Suspense: 更先进的组件

### Vue3.0变得更快的原理

#### Diff方法的优化

 Vue2中的虚拟dom是进行全量比对

 Vue3新增了静态标记（**PatchFlag**）  
在与上次虚拟节点进行比对时候，只对比带有patch flag的节点  
并且可以通过flag的信息 得知当前节点要比对的具体内容

**附录PatchFlag：**

export const enum PatchFlags {

TEXT = 1, // 动态文本节点

CLASS = 1 << 1, // 2 动态class

STYLE = 1 << 2, // 4 动态style

PROPS = 1 << 3, // 8 动态属性，但不包含类名和样式

FULL\_PROPS = 1 << 4, // 16 具有动态 key 属性，当 key 改变时，需要进行完整的 diff 比较

HYDRATE\_EVENTS = 1 << 5, // 32 带有监听事件的节点

STABLE\_FRAGMENT = 1 << 6, // 64 一个不会改变子节点顺序的 fragment

KEYED\_FRAGMENT = 1 << 7, // 128 带有key属性的 fragment 或部分带有 key

UNKEYED\_FRAGMENT = 1 << 8, // 256 子节点没有 key 的 fragment

NEED\_PATCH = 1 << 9, // 512 一个节点只会进行非 props 比较

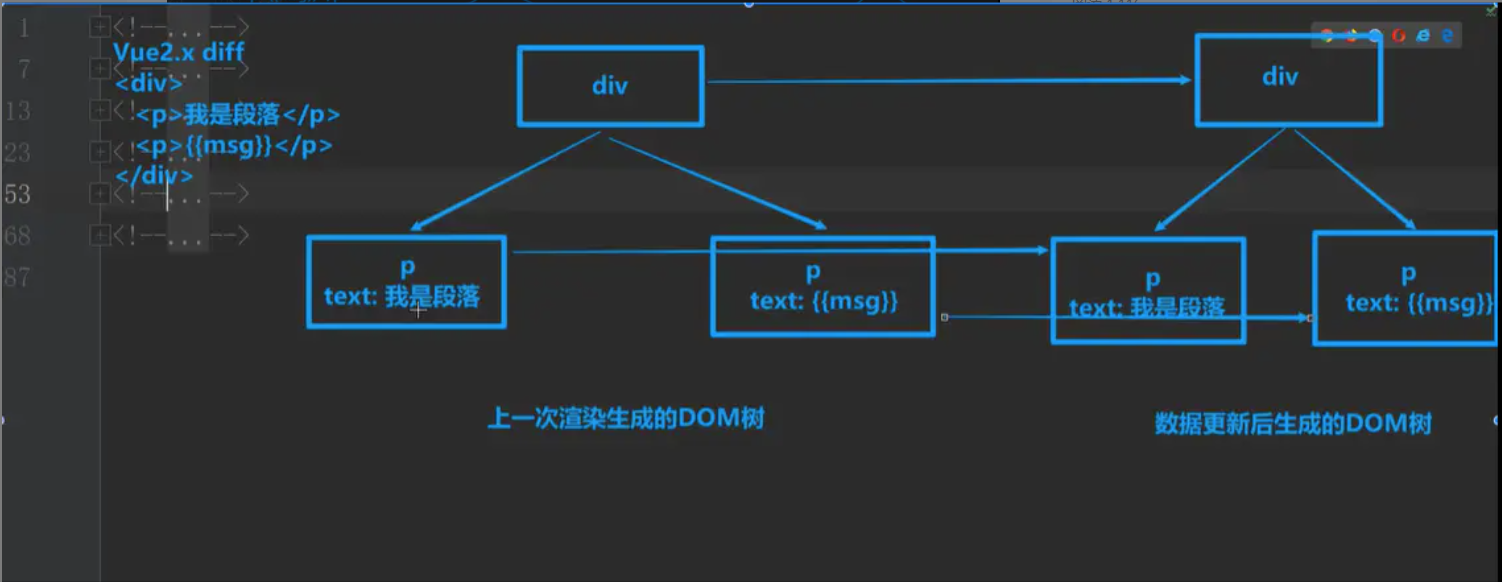
DYNAMIC\_SLOTS = 1 << 10, // 1024

HOISTED = -1,

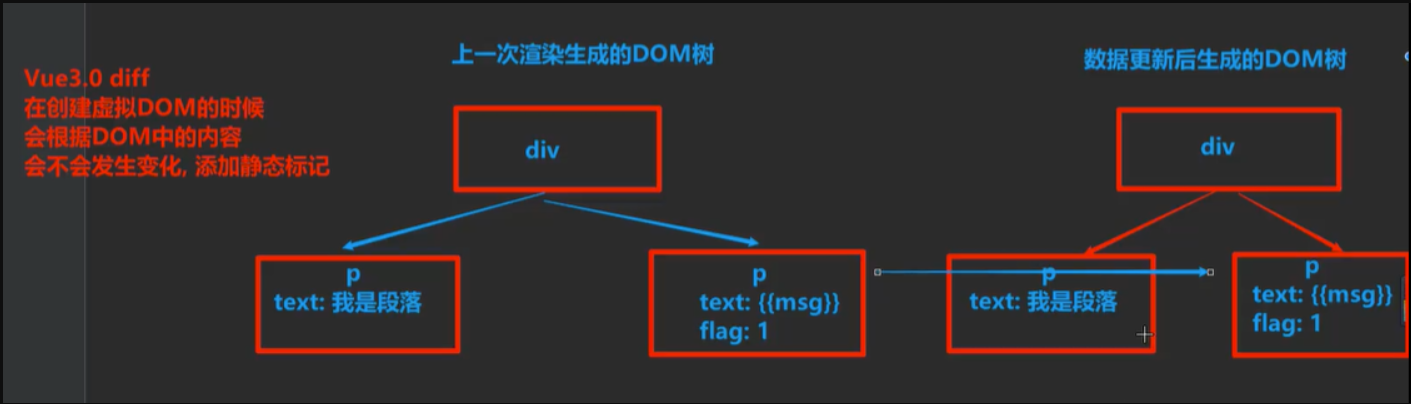
BAIL = -2

}

##### Vue2 diff算法



##### Vue3.0diff算法



##### Vue编译dom原理

<div>

<p>我是段落</p>

<p>{{msg}}</p>

</div>

import {

createVNode as \_createVNode,

toDisplayString as \_toDisplayString,

openBlock as \_openBlock,

createBlock as \_createBlock

} from "vue"

export function render(\_ctx, \_cache, $props, $setup, $data, $options) {

return (\_openBlock(), \_createBlock("div", null, [

\_createVNode("p", null, "我是段落"),

\_createVNode("p", null, \_toDisplayString(\_ctx.msg), 1 /\* TEXT \*/)

]))

}

##### 附录PatchFlags

export const enum PatchFlags {

TEXT = 1, // 动态文本节点

CLASS = 1 << 1, // 2 动态class

STYLE = 1 << 2, // 4 动态style

PROPS = 1 << 3, // 8 动态属性，但不包含类名和样式

FULL\_PROPS = 1 << 4, // 16 具有动态 key 属性，当 key 改变时，需要进行完整的 diff 比较

HYDRATE\_EVENTS = 1 << 5, // 32 带有监听事件的节点

STABLE\_FRAGMENT = 1 << 6, // 64 一个不会改变子节点顺序的 fragment

KEYED\_FRAGMENT = 1 << 7, // 128 带有key属性的 fragment 或部分带有 key

UNKEYED\_FRAGMENT = 1 << 8, // 256 子节点没有 key 的 fragment

NEED\_PATCH = 1 << 9, // 512 一个节点只会进行非 props 比较

DYNAMIC\_SLOTS = 1 << 10, // 1024

HOISTED = -1,

BAIL = -2

}

#### 静态提升（hoist Static）

* Vue2中无论元素是否参与更新，每次都会重新创建，然后再渲染
* Vue3中对于不参与更新的元素，会做静态提升，只会被创建一次，在渲染时直接复用即可

##### Vue编译demo

<div>

<p>我是段落</p>

<p>{{msg}}</p>

</div>

import {

createVNode as \_createVNode,

toDisplayString as \_toDisplayString,

openBlock as \_openBlock,

createBlock as \_createBlock

} from "vue"

const \_hoisted\_1 = /\*#\_\_PURE\_\_\*/\_createVNode("p", null, "我是段落", -1 /\* HOISTED \*/)

export function render(\_ctx, \_cache, $props, $setup, $data, $options) {

return (\_openBlock(), \_createBlock("div", null, [

\_hoisted\_1,

\_createVNode("p", null, \_toDisplayString(\_ctx.msg), 1 /\* TEXT \*/)

]))

}

#### 事件侦听器缓存（擦车Handlers）

* 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

##### Vue编译demo

<div>

<button @click="onClick">按钮</button>

</div>

关闭侦听器缓存

import {

createVNode as \_createVNode,

openBlock as \_openBlock,

createBlock as \_createBlock

} from "vue"

export function render(\_ctx, \_cache, $props, $setup, $data, $options) {

return (\_openBlock(), \_createBlock("div", null, [

\_createVNode("button", { onClick: \_ctx.onClick }, "按钮", 8 /\* PROPS \*/, ["onClick"])

]))

}

开启事件侦听器缓存

import {

createVNode as \_createVNode,

openBlock as \_openBlock,

createBlock as \_createBlock

} from "vue"

export function render(\_ctx, \_cache, $props, $setup, $data, $options) {

return (\_openBlock(), \_createBlock("div", null, [

\_createVNode("button", {

onClick: \_cache[1] || (\_cache[1] = (...args) => (\_ctx.onClick(...args)))

}, "按钮")

]))

}

**注意**：侦听器缓存是根据静态标记进行的，只有有静态标记的才会进行比较，才会进行跟踪

#### Ssr渲染

 当有大量静态内容时候，这些内容会被当做纯字符串推进一个buffer里面，  
即使存在动态绑定，会通过模板插值嵌入进去。这样会比通过虚拟dom来渲染的快上很多很多。

 当静态内容大到一定量级时候，会用\_createStaticVNode方法在客户端生成一个static node，这些静态node，会被直接innerHTML，就不需要创建对象，然后根据对象渲染。