

用得上的Vue3的经验/技巧

我要充满正能量 2023-05-26 👁4,897 ⌚阅读20分钟

关注

用得上的Vue3的经验/技巧

本文会持续更新,在vue3开发中遇到的问题和总结

🔔提示:本文需要初步了解Vue3相关基础,为方便未使用TS

🤖GPT🤖的出现已经大大改变了我的开发习惯,从以前的**面向搜索引擎编程**,到现在的**面向GPT编程**,本文内容,我将会采用GPT一问一答的形式,然后对GPT的回答进行解释和补充;如有自己观点欢迎评论区给出指正,互相学习~🦀🦀🦀🦀🦀🦀🦀

1. 开发Vue3.x一定要用TS吗?

1.1. 🤖GPT🤖: 不一定需要使用

不一定需要使用 TypeScript (TS) 来开发 Vue 3 应用程序,但是使用 TypeScript 可以提供一些优点。

Vue 3 支持使用原生 JavaScript 编写应用程序,因此你可以选择使用 JavaScript 进行开发。然而,使用 TypeScript 可以提供更好的类型检查和代码提示,以及更好的代码可读性和可维护性。TypeScript 还可以帮助你避免一些常见的 JavaScript 错误,例如类型错误和拼写错误。

除此之外,Vue 3 的官方文档和许多社区资源都提供了 TypeScript 的示例和文档,因此使用 TypeScript 可以更轻松地在 Vue 3 生态系统中找到帮助和支持。

综上所述,使用 TypeScript 开发 Vue 3 应用程序可以提供更好的开发体验和更好的代码质量,但如果你更喜欢使用原生 JavaScript,也是完全可以的。

1.2. 🙌我🙌: 推荐使用

推荐使用,缺点无非就是学习成本高,学习有多难呢?无非就是大家说的卷点,但是TS的强类型语言特性,如果学会了对学习如java等强类型语言是大有益处了,并且github上有很多优秀的vue3后台管理系统项目基本采用TS(搜索关键字:vue3 admin)

Vue3本身就是用TS重构的,Vue3有较好的TS支持的,比如**类型检查**和**代码提示**,开发Vue3使用不使用TS,其实很多方面的限制,团队成员的技术栈,成员会不会TS, 如:①学习成本高②大量的anyscript用户~~~

如果习惯了TS提供的类型检查和代码提示,跳转到定义(ctrl+鼠标左键)就能找到源头的感觉,就真回不去了



javascript 复制代码

```
<script setup lang="ts">
import { ref } from "vue";
//声明一个ref响应式变量,类型限制为Boolean
let demo = ref<Boolean>(true);
//当给这个变量赋值类型不是Boolean的时候就会标红报错
demo.value = 1;
</script>
```

类型检查如下:

当给demo这个类型限定为Boolean的变量赋值为其他基础类型的数据时,类型检查触发,爆红提示

不能将类型“number”分配给类型“{ valueOf: () => boolean; }”。 ts(2322)

```
const demo: Ref<{
  valueOf: () => boolean;
}>
```

[查看问题 \(Alt+F8\)](#) 没有可用的快速修复

demo.value = 1;

@稀土掘金技术社区

2. ref 声明的响应式变量访问和修改都得加 .value 小尾巴有点麻烦怎么办?

老老实实写,目前没好的方式,但是volar插件支持补全!贼好用!记得开起来!!!!

2.1. 🤖 GPT 🤖: 通过computed计算属性解决(代码冗余,不好)

在使用 `ref` 声明响应式数据时,访问和修改数据确实需要加上 `.value`,但是你可以使用 `computed` 属性来简化这个过程。

`computed` 属性可以将响应式数据转换为计算属性,使其可以像普通属性一样访问和修改,而无需每次都加上 `.value`。下面是一个例子:

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

let count = ref(0)

// 将响应式数据转换为计算属性
const countValue = computed({
  get: () => count.value,
  set: (val) => count.value = val
})

// 访问和修改计算属性时就不需要加上 .value 了
console.log(countValue) // 输出 0
countValue++ // 等价于 count.value++
```

javascript 复制代码

使用 `computed` 属性可以简化代码,并且更符合 Vue 的响应式数据处理方式。

2.2. 🙌 我 🙌: 2种方案

GPT的答案,确实解决了不用频繁了 `.value` 的问题,但是通过computed计算属性来实现,代码更加冗余没啥现实意义;(太繁琐!麻烦!)

1. `$()`响应式语法糖(已废弃)

Vue3.x有个版本提供了一种实验性语法糖解决 `ref` 创建的响应式变量读取和修改时,频繁 `.value` 的问题;

该提案在2023-2的时候由尤大自己废弃了 🗑️🗑️🗑️🗑️ (github.com)

yyx990803 on Feb 21

Maintainer

作者

...

正如你们中的许多人已经知道的那样，我们在团队的共识下正式放弃了这个 RFC。

The Rationale

反应性转换的最初目标是通过在处理反应状态时提供更简洁的语法来改善开发人员体验。我们将其作为实验性发布，以收集来自现实世界使用情况的反馈。尽管提出了好处，但我们发现了以下问题：

1. 失去更难分辨正在跟踪的内容以及哪条线正在触发反应效应。这个问题在小型 SFC 中并不明显，但在大型代码库中，心理开销变得更加明显，特别是如果语法也在 SFC 之外使用。 `.value`
2. 由于 (1)，一些用户选择仅在 SFC 中使用反应性转换，这造成了不同心智模型之间上下文转换的不一致和成本。因此，困境在于仅在 SFC 内部使用它会导致不一致，但在 SFC 外部使用它会损害可维护性。
3. 由于仍然会有外部函数期望使用原始引用，因此反应变量和原始引用之间的转换是不可避免的。这最终增加了一件需要学习的东西和额外的精神负担，我们注意到这比普通的作曲 API 更让初学者感到困惑。

最重要的是，潜在的碎片化风险。尽管这显然是选择加入的，但一些用户对该提案表示强烈反对，原因是他们担心他们将不得不使用不同的代码库，其中一些人选择使用它，而另一些人则没有。这是一个合理的问题，因为反应性转换需要一种不同的心智模型来扭曲 JavaScript 语义（变量赋值能够触发反应效应）。

考虑到所有因素，我们认为将其作为稳定功能将导致更多的问题而不是收益，因此不是一个好的权衡。

Migration Plans

- 该功能已经通过 `Vue` 宏以外部包的形式得到支持。
- 3.3: 该功能将被标记为已弃用。它将继续工作，但在此期间你应该迁移到 `Vue` 宏。
- 3.4: 该功能将从核心中删除，除非使用 `Vue` 宏，否则将不再工作。

↑ 17

👍 95

👎 9

🗨️ 1

👤 13

@稀土掘金技术社区

总结废弃原因就是,普罗大众都反对,只能废弃;(很显然,官方也对频繁.value深恶痛绝,但是又因为一些原因去掉了这个语法糖;🙄🙄🙄)

\$ 响应式语法糖来解决 `ref` 响应式属性频繁 `.value` 的问题(已废弃❌❌❌):

该语法糖是一种编译宏,在被 `$()` 标记后的变量会在编译阶段自动加入 `.value`,也就是编译后的代码也是有 `.value` 的

javascript 复制代码

```
<script setup>
import { ref } from 'vue'

const loadingRef = ref(true)
//通过$( )将ref响应式变量
let loading = $(loadingRef)
console.log(loadingRef.value === $(loadingRef) === loading)
console.log(loadingRef === $(loading))
</script>
```

废弃的原因很多,大部分人认为,分不清是正常变量还是响应式变量(`.value` 能一眼看出是 `ref` 声明的响应式变量)

```

<script setup>
import { ref } from 'vue'
//常量
const constLoading = true

//普通ref声明的响应式变量
const loadingRef = ref(true)
//loadingRef.value 使用 .value你能很清晰知道 loadingRef是一个ref的响应式变量
console.log(loadingRef.value)

//通过$()将ref响应式变量
let loading = $(loadingRef)
// loading虽然能直接使用,不需要.value, 但是如果代码非常长,你就可能忘记 loading是一个响应式变量
console.log(loading)
//如果在review代码过程, 你根本不知道 constLoading 和 loading 哪个是响应式变量
</script>

```

从上面的例子看,如果在review代码过程, 你根本不知道 `constLoading` 和 `loading` 哪个是响应式变量,但是 `loadingRef` 带的 `.value` 小尾巴你就能很清楚知道这是一个响应式变量;

这里推荐一篇文章里面讲了:🐞🐞🐞🐞 [\\$响应式语法糖的废弃原因](#)

2. 使用reactive替代ref(不推荐)

下面关于reactive和ref使用哪个的问题,有具体分析;推荐使用ref,不要惧怕 `.value` ;因为有插件能自动补充!!!而且 `.value` 你一眼就能看出是响应式变量.而reactive却不明显;

有些人喜欢用 `reactive` 替代 `ref` ,像option API 一样将响应式数据放入data对象中

```

<script setup>
let data = reactive({
  a:1,
  b:2
})

```

```
  })  
</script>
```

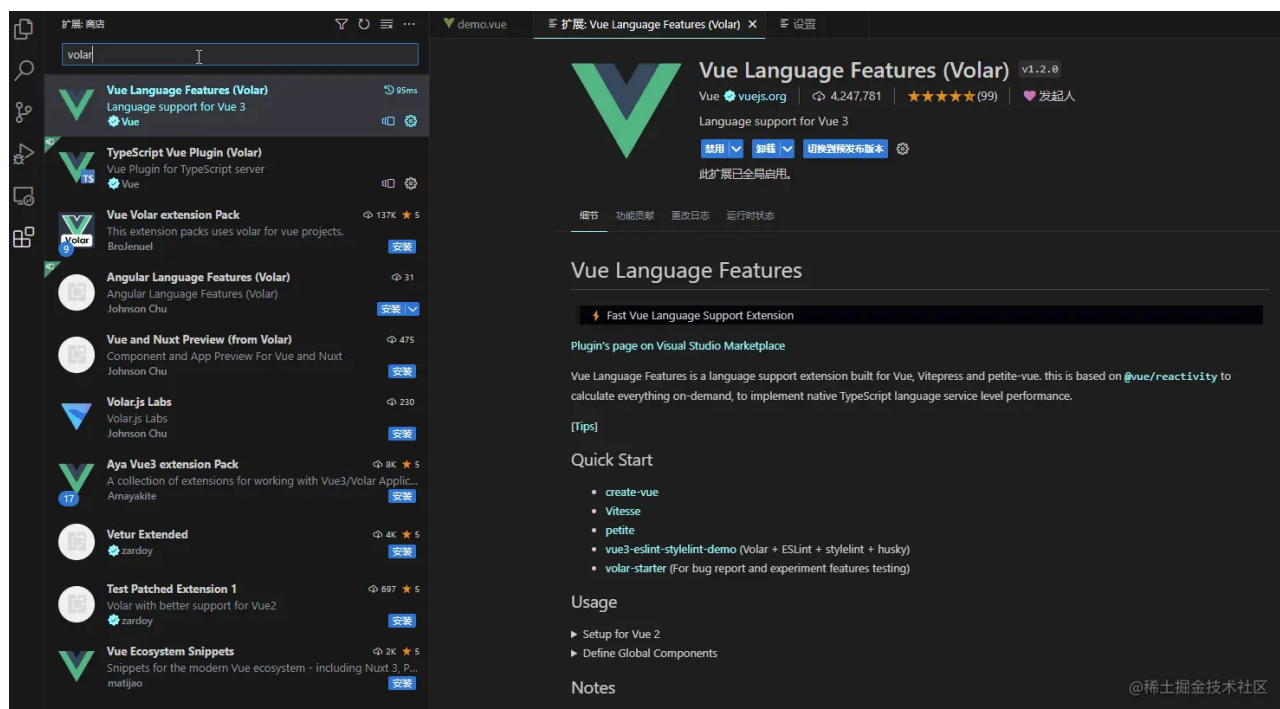
reactive存在一些问题:

- 使用不当容易失去响应(下面有介绍)
- 只能声明引用数据类型,基本数据类型都需要封装到对象里面

3. volar插件能自动补全 `.value` (强烈推荐!!!!!!)

本人推荐ref一把梭,但是ref又得到处 `.value` ,那就交给插件来完成吧!!!

- `valor` 自动补全 `.value` (不是默认开启,需要手动开启)
- 不会有人不知道Vue3需要不能使用 `vetur` 要用 `valor` 替代吧?不会不会吧? (必备volar插件)



- 可以看到当输入ref声明的响应式变量时,volar插件自动填充 `.value` 那还有啥烦恼呢? 方便!

4 使用unref()读取ref

该方法仅限在读取的时候使用,如果需要修改还是需要 `.value`

- `unref()` 传入`ref`类型的响应式参数,返回`ref`响应数据去掉 `.value` 的值
- `unref()` 传入普通参数, 原样返回
- 这是一个语法糖: `val = isRef(val) ? val.value : val`

利用该属性可以在读取`ref`类型的响应式数据时不用频繁 `.value`



scss 复制代码

```
//读取Element-Plus 中form组件的resetFields方法
let elFormRef = ref()
//通过unref
unref(elFormRef)?.resetFields()
//通过.value
elFormRef.value?.resetFields()
```

3.ref和reactive用哪个?

3.1. 🤖GPT🤖:ref和reactive用哪个根据数据类型而定

`ref` 用于将基本类型的数据（如字符串、数字、布尔值等）转换为响应式数据。使用 `ref` 定义的数据可以通过 `.value` 属性访问和修改。

`reactive` 用于将对象转换为响应式数据，包括复杂的嵌套对象和数组。使用 `reactive` 定义的数据可以直接访问和修改属性。

因此，根据使用场景的不同，选择使用 `ref` 还是 `reactive` 是有区别的：

- 如果你只需要处理**基础类型**的数据，例如字符串、数字、布尔值等，可以使用 `ref`。
- 如果你需要处理**复杂的嵌套对象或数组**，可以使用 `reactive`。

当然，在实际开发中，你也可以同时使用 `ref` 和 `reactive`，根据需要进行选择相应的 API 进行操作。例如，可以使用 `ref` 来处理单个数据，而使用 `reactive` 来处理包含多个数据的对象或数组。

3.2. 🙌我🙌:推荐ref一把梭

reactive 和 ref 对比

reactive

✗ 只支持对象和数组(引用数据类型)

✓ 在 `<script>` 和 `<template>` 中无差别使用

✗ 重新分配一个新对象会丢失响应性

能直接访问属性

✗ 将对象传入函数时,失去响应

✗ 解构时会丢失响应性,使用toRefs

ref

✓ 支持基本数据类型+引用数据类型

✗ 在 `<script>` 和 `<template>` 使用方式不同(script中要 `.value`)

✓ 重新分配一个新对象**不会**失去响应

需要使用 `.value` 访问属性

✓ 传入函数时,不会失去响应

✗ 解构对象时会丢失响应性,使用toRefs

3.2.1前提知识:

3.2.1.1. ref既能声明基本数据类型,也能声明对象和数组;

Vue 提供了一个 `ref()` 方法来允许我们创建可以使用**任何值类型**的响应式 `ref`

javascript 复制代码

```
//对象
const state = ref({})

//数组
const state2 = ref([])
```

3.2.1.2. reactive失去响应的情况:

`reactive` 一时爽,使用不恰当的时候失去响应泪两行,开开心心敲代码过程中,会感叹!!咦?怎么不行?为什么这么赋值失去响应了? 辣鸡reactive!!! 我要用 `ref` 🙌🙌
yyds

1. 给响应式对象的字面量赋一整个普通对象/reactive对象

通常在页面数据回显时,需要将AJAX请求获取的对象直接赋值给响应式对象,如果操作不当就导致 **reactive**声明的对象 失去响应

- 赋值一个普通对象



javascript 复制代码

```
let state = reactive({ count: 0 })
//这个赋值将导致state失去响应
state = {count: 1}
```

- 赋值一个 **reactive** 对象

如果给reactive的响应式对象赋值普通对象会失去响应,那么给它赋值一个reactive的响应式对象不就行了吗?下面试试看



html 复制代码

```
<template>
  {{state}}
</template>

<script setup>
const state = reactive({ count: 0 })
//nextTick异步方法中修改state的值
nextTick(() => {
  //并不会触发修改DOM ,说明失去响应了
  state = reactive({ count: 11 });
});
</script>
```

在 **nextTick** 中给 **state** 赋值一个reactive的响应式对象,但是DOM并没有更新!

解决方法:

1. 不要直接整个对象替换,对象属性一个个赋值



js 复制代码

```
let state = reactive({ count: 0 })
//state={count:1}
```

```
state.conut = 1
```

2. 使用ref定义对象

个人推荐,非必要不用reactive

js 复制代码

```
let state = ref({ count: 0 })
state.value={count:1}
```

2.将响应式对象的属性-赋值给变量(断开连接/深拷贝)

这种类似深拷贝不共享同一内存地址了,只是字面量的赋值;对该变量赋值也不会影响原来对象的属性值

javascript 复制代码

```
let state = reactive({ count: 0 })
//赋值
// n 是一个局部变量, 同 state.count
// 失去响应性连接
let n = state.count
// 不影响原始的 state
n++
console.log(state.count) //0
```

有人就说了,既然赋值对象的属性,那我赋值一整个对象不就是浅拷贝了吗?那不就是上面说的 给响应式对象的字面量赋一整个普通对象/reactive对象 这种情况吗?这种是会失去响应的

3.直接解构时

- 直接解构会失去响应

解构ref声明的响应式对象也会失去响应

javascript 复制代码

```
let state = reactive({ count: 0 })
//普通解构count 和 state.count 失去了响应性连接
let { count } = state
count++ // state.count值依旧是0
```

- 使用 `toRefs` 解构不会失去响应

javascript 复制代码

```
const state = reactive({ count: 0 })
//使用toRefs解构
let { count } = toRefs(state)
count++ // state.count值改变为1
```

4.传入一个函数时

将reactive声明的响应式对象的参数传入一个函数时,会失去响应

javascript 复制代码

```
let state = reactive({ count: 0 })

const fn = (count) => {
  // ⚠️ 这里的 count 是一个普通的 number (不是)
  console.log(count)
  // 不能跟踪 state.count 的改变
}

//调用该函数,传入响应式对象的参数
fn(state.count)
```

3.2.2. ref一把梭

当使用reactive时,如果不了解**reactive失去响应的情况**,那么使用reactive会造成很多困扰!

推荐使用 `ref` 总结原因如下:

0. reactive只能声明引用数据类型(对象/数组)

1. reactive在一些情况下会失去响应,这个情况会导致数据回显失去响应(数据改了,dom没更新)

给响应式对象的字面量赋一整个普通对象,将会导致reactive声明的响应式数据失去响应

javascript 复制代码

```
<template>
  {{state.a}}
  {{state.b}}
  {{state.c}}
</template>

<script>
let state = reactive({ a:1,b:2,c:3 })
onMounted(()=>{
  //通AJAX请求获取的数据,回显到reactive,如果处理不好将导致变量失去响应,
  //回显失败,给响应式数据赋值一个普通对象
  state = { a:11,b:22,c:333 }
  //回显成功,一个个属性赋值
  state.a = 11
  state.b = 22
  state.c = 33
})
</script>
```

csharp 复制代码

上面这个例子如果是使用ref进行声明,直接赋值即可,不需要将属性拆分一个个赋值

使用ref替代reactive:

javascript 复制代码

```
<template>
  {{state.a}}
  {{state.b}}
  {{state.c}}
</template>

<script>
let state = ref({ a:1,b:2,c:3 })
onMounted(()=>{
```

```
//回显成功
state.value = { a:11,b:22,c:333 }
})
</script>
```

2. ref适用范围更大,声明的数据类型.基本数据类型和引用数据类型都行

虽然使用ref声明的变量,在读取和修改时都需要加 `.value` 小尾巴,但是正因为是这个小尾巴,我们review代码的时候就很清楚知道这是一个ref声明的响应式数据;

4. 为什么要用setup语法糖? `<script setup>`

关于setup语法糖的内容你可以看我的另一篇文章:[Vue3拒绝写return, 用setup语法糖,让写Vue3更畅快 - 掘金 \(juejin.cn\)](#)

4.1 🤖GPT🤖:

Vue 3中引入了 `setup` 语法糖,旨在提供一种新的、更加简洁、灵活的组件写法,以及提高性能和开发体验。以下是使用 `setup` 语法糖的好处:

0. 更加简洁明了 使用 `setup` 语法糖,可以将组件的选项(如data、methods等)都统一放在一个函数中,使代码更加简洁明了。同时,由于 `setup` 函数中不需要使用 `this` 关键字,避免了 `this` 指向混乱的问题。
1. 更加灵活 使用 `setup` 语法糖可以更加灵活地组合和重用逻辑。`setup` 函数可以返回一个对象,使得我们可以使用解构赋值和对象合并来组合多个逻辑。同时,由于 `setup` 函数是一个普通的JavaScript函数,因此可以使用任何JavaScript语法和库来实现我们想要的逻辑。
2. 更好的类型推断和静态分析 `setup` 语法糖在类型推断和静态分析方面也优于Vue 2.x中的选项式API。由于 `setup` 函数的返回值可以被编译器推断为一个包含响应式数据和方法的对象,因此在开发过程中可以获得更好的类型检查和自动补全。
3. 更好的性能 由于 `setup` 函数中的响应式数据和方法都是在创建组件实例时进行初始化的,因此可以在组件实例化时完成响应式数据的初始化,从而减少了组件的初始化时间和内存开销,提高了性能。

综上所述, `setup` 语法糖提供了一种更加简洁、灵活、可维护的组件写法,并且在类型推断和静态分析方面也更加优秀。因此在Vue 3中,推荐使用 `setup` 语法糖来编写组件。

4.2. 🙌我🙌:简化代码

GPT的回答较为抽象,setup语法糖,所谓语法糖就是一种简便的写法,将复杂繁琐的写法变的简单:

1. `<script setup>` 中无需return 声明的变量、函数以及import引入的内容, 即可在 `<template/>` 使用

- `<script setup>` 语法糖



html 复制代码

```
<script setup>
//import引入的内容
import { getToday } from './utils'
// 变量
const msg = 'Hello!'
// 函数
function log() {
  console.log(msg)
}
</script>

//在template中直接使用声明的变量、函数以及import引入的内容
<template>
  <div @click="log">{{ msg }}</div>
  <p>{{getToday()}}</p>
</template>
```

- 标准组件 `<script>` 需要写setup函数并繁琐return



html 复制代码

```
<script>
//import引入的内容
import { getToday } from './utils'
export default{
  setup(){
    // 变量
    const msg = 'Hello!'
    // 函数
    function log() {
      console.log(msg)
    }
    //想在template里面使用需要在setup内return暴露出来
    return{
```

```

      msg,
      log,
      getToday
    }
  }
}
</script>

<template>
  <div @click="log">{{ msg }}</div>
  <p>{{getToday()}}</p>
</template>

```

总结: `<script setup>` 语法糖里面的代码会被编译成组件 `setup()` 函数的内容, 不需要通过`return`暴露 声明的变量、函数以及`import`引入的内容, 即可在 `<template/>` 使用, 并且**不需要写 `export default {}`

`<script setup>` 语法糖里面的代码会被编译成组件 `setup()` 函数的内容。这意味着与普通的 `<script>` 只在组件被首次引入的时候执行一次不同, `<script setup>` 中的代码会在**每次组件实例被创建的时候执行



javascript 复制代码

```

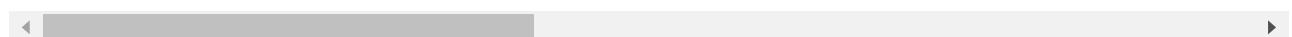
<script>
  console.log('script');//多次实例组件, 只触发一次
  export default {
    setup() {
      console.log('setupFn');//每次实例化组件都触发和script-setup标签一样
    }
  }
</script>

```



arduino 复制代码

(`script-setup`标签最终都会编译成``setup()`` 函数的内容, 每次实例化组件, 就是实例化一次`setup`函数。`script-`



2、`<script setup>` 将自动注册引入组件

不需要在引入组件后, 通过 `components: {}` 注册组件, 可直接使用


```
<script setup>
import MyComponent from './MyComponent.vue'
//components:{MyComponent} 不需要注册直接使用
</script>

<template>
  <MyComponent />
</template>
```

总结:就是为了简化之前的 `setup`函数 内,频繁 `retrun` 的问题

5 为什么要用自定义Hook?

关于这个问题可以看我的另外一篇文章:[Vue3必学技巧-自定义Hooks-让写Vue3更畅快 - 掘金 \(juejin.cn\)](#)

5.1 🤖GPT🤖:

Vue 3中引入了自定义Hook, 旨在提供一种可复用的逻辑, 以减少重复代码, 提高代码重用性和可维护性。以下是使用自定义Hook的好处:

0. 提高代码的可复用性 自定义Hook可以将通用的逻辑封装成可复用的模块, 从而减少了代码的重复, 提高了代码的复用性。
1. 更好的组织和维护代码 使用自定义Hook可以将组件中的逻辑分离出来, 使得代码更加清晰易懂。同时, 自定义Hook也可以使得组件的代码更加简洁, 只需要使用自定义Hook来获取需要的数据或执行需要的操作即可。这样可以降低代码耦合度, 从而提高代码的可维护性。
2. 更好的测试性 自定义Hook使得测试更加容易, 因为我们可以单独测试自定义Hook中的逻辑, 而不需要依赖组件或其他外部依赖。这样可以提高代码的可测试性, 从而提高代码的质量和可维护性。
3. 更好的类型推断和静态分析 自定义Hook可以使用TypeScript等类型检查工具进行类型检查和静态分析, 从而提高代码的质量和可维护性。

综上所述, 自定义Hook是Vue 3中的一个重要特性, 它可以使得我们更加方便地复用逻辑, 提高代码重用性和可维护性。因此, 在Vue 3中, 推荐使用自定义Hook来组织复杂逻辑。

辑，并提高代码质量和可维护性。

5.2. 🙌我🙌: Vue3悄咪咪说,React的Hooks用过都说好,我要学过来

Vue3中的自定义Hooks其实很多程度上是借鉴了React的

5.2.1. 怎么创建自定义HOOK?

步骤:

0. 将可复用功能抽离为外部JS文件
1. 函数名/文件名以use开头，形如： `useXX`
2. 引用时将响应式变量或者方法显式暴露出来如： `const {nameRef, Fn} = useXX()`

(在setup函数解构出自定义hooks的变量和方法)

创建自定义Hook实例:

简单的加减法计算，将加法和减法抽离为2个自定义Hooks，并且相互传递响应式数据

- 加法功能-Hook

`useAdd.js` 由 `export default`导出

javascript 复制代码

```
import { ref, watch } from 'vue';
const useAdd = ({ num1, num2 }) => {
  const addNum = ref(0)
  watch([num1, num2], ([num1, num2]) => {
    addFn(num1, num2)
  })
  const addFn = (num1, num2) => {
    addNum.value = num1 + num2
  }
  return {
```

```
        addNum,  
        addFn  
      }  
    }  
    //默认导出Hook  
    export default useAdd
```

- 减法功能-Hook

useSub.js 由export 导出



javascript 复制代码

```
//减法功能-Hook  
import { ref, watch } from 'vue';  
//导出Hook  
export function useSub ({ num1, num2 }){  
  const subNum = ref(0)  
  watch([num1, num2], ([num1, num2]) => {  
    subFn(num1, num2)  
  })  
  const subFn = (num1, num2) => {  
    subNum.value = num1 - num2  
  }  
  return {  
    subNum,  
    subFn  
  }  
}
```

- 加减法计算组件



javascript 复制代码

```
<template>  
  <div>  
    num1:<input v-model.number="num1" style="width:100px" />  
    <br />  
    num2:<input v-model.number="num2" style="width:100px" />  
  </div>  
  <span>加法等于:{{ addNum }}</span>  
  <br />  
  <span>减法等于:{{ subNum }}</span>  
</template>
```

```

<script setup>
import { ref } from 'vue'
import useAdd from './useAdd.js' //引入自定义hook
import { useSub } from './useSub.js' //引入自定义hook

const num1 = ref(2)
const num2 = ref(1)
//加法功能-自定义Hook (将响应式变量或者方法形式暴露出来)
const { addNum, addFn } = useAdd({ num1, num2 })
addFn(num1.value, num2.value)
//减法功能-自定义Hook (将响应式变量或者方法形式暴露出来)
const { subNum, subFn } = useSub({ num1, num2 })
subFn(num1.value, num2.value)
</script>

```

5.2.2. Vue3引入自定义Hooks替代Mixin,优化了Mixin的一些弊端:

1、Mixin难以追溯的方法与属性！Vue3自定义Hooks却可以

Vue3自定义Hooks, 引用时将响应式变量或者方法显式暴露出来如：`const {nameRef, Fn} = useXX()`

- Mixins



javascript 复制代码

```

export default {
  mixins: [ a, b, c, d, e, f, g ], //一个组件内可以混入各种功能的Mixin
  mounted() {
    console.log(this.name) //问题来了，这个name是来自于哪个mixin?
  }
}

```



复制代码

Mixin不明的混淆，我们根本无法获知属性来自于哪个Mixin文件，给后期维护带来困难

- Vue3自定义Hooks



javascript 复制代码

```
//加法功能-自定义Hook (将响应式变量或者方法形式暴露出来)
const { addNum, addFn } = useAdd({ num1, num2 })
addFn(num1.value, num2.value)
//减法功能-自定义Hook (将响应式变量或者方法形式暴露出来)
const { subNum, subFn } = useSub({ num1, num2 })
subFn(num1.value, num2.value)
```

我们很容易看出每个Hooks显式暴露出来的响应式变量和方法

2、无法向Mixin传递参数来改变逻辑，但是Vue3自定义Hooks却可以：

Vue3自定义Hooks可以灵活传递任何参数来改变它的逻辑，参数不限于其他hook的暴露出来的变量

- Mixins

javascript 复制代码

```
export default {
  mixins: [ addMixin, subMixin], //组件内混入加法和减法Mixin
  mounted(){
    this.add(num1,num2) //调用addMixin内部的add方法
    this.sub(num1,num2) //调用subMixin内部的sub方法
  }
}
```

可以通过调用Mixin内部方法来传递参数，却无法直接给Mixin传递参数，因为Mixin不是函数形式暴露的，不发传参

- Vue3自定义Hook

在上面实例基础上添加个算平均的Hook

javascript 复制代码

```
//平均功能-Hook
import { ref, watch } from "vue";
export function useAverage(addNum) {
  const averageNum = ref(0);
  watch(addNum, (addNum) => {
```

```

    averageFn(addNum);
  });
  const averageFn = (addNum) => {
    averageNum.value = addNum / 2;
  };
  return {
    averageNum,
    averageFn,
  };
}

```

组件内



javascript 复制代码

```

//组件内
//加法功能-自定义Hook（将响应式变量或者方法形式暴露出来）
const { addNum, addFn } = useAdd({ num1, num2 })
addFn(num1.value, num2.value)//主动调用，返回最新addNum
//平均功能-自定义Hook- hook传入参数值来其他hook暴露出来的变量
const { averageNum, averageFn } = useAverage(addNum)
averageFn(addNum.value)

```



复制代码

Vue3自定义Hooks可以灵活传递任何参数来改变它的逻辑，参数不限于其他hook的暴露出来的变量，这提高了Vue3在抽



3、Mixin同名变量会被覆盖，Vue3自定义Hook可以在引入的时候对同名变量重命名

- Mixins



javascript 复制代码

```

export default {
  mixins: [ addMixin, subMixin], //组件内混入加法和减法Mixin
  mounted(){
    this.add(num1,num2) //调用加法addMixin内部的add方法
    this.sub(num1,num2) //调用减法subMixin内部的sub方法
  }
}

```

如果 `this.add(num1,num2)` 和 `this.sub(num1,num2)` 计算的结果返回的同名变量 `totalNum`，由于JS单线程，后面引入的会覆盖前面的，`totalNum`最终是减法`sub`的值

- Vue3自定义Hooks

javascript 复制代码

```
//加法功能-自定义Hook（将响应式变量或者方法形式暴露出来）
const { totalNum:addNum, addFn } = useAdd({ num1, num2 })
addFn(num1.value, num2.value)
//减法功能-自定义Hook（将响应式变量或者方法形式暴露出来）
const { totalNum:subNum, subFn } = useSub({ num1, num2 })
subFn(num1.value, num2.value)
```

在Vue3自定义Hooks中，虽然加法和减法Hooks都返回了`totalNum`，但是利用ES6对象解构很轻松给变量重命名

6. Composition Api 和option Api 用哪个?

vue3.x中是兼容option Api 写法的

6.1 🤖GPT🤖: 看喜好

在Vue 3中，我们可以使用两种不同的API来编写组件：Composition API和Options API。两种API各有优缺点，选择哪种API主要取决于个人偏好和项目需求。

Options API是Vue 2.x中的常用API，它通过一个包含各种选项的对象来描述组件。每个选项都有自己的作用，例如 `data` 选项用于定义组件的响应式数据，`methods` 选项用于定义组件的方法等。Options API的优点是易于理解和上手，适合编写小型组件，也可以快速地实现一些简单的功能。

Composition API是Vue 3中新增的API，它通过一组函数来描述组件，可以更加灵活地组织组件的逻辑。Composition API的优点是提供了更好的代码组织和复用性，可以更好地处理组件的复杂逻辑，而且可以更容易地进行单元测试。因此，对于较大型的组件和复杂的业务逻辑，建议使用Composition API来编写组件。

当然，对于一些小型的组件或者对Vue 2.x较为熟悉的开发者，仍然可以使用Options API来编写组件。在实际项目开发中，也可以根据项目需求和个人偏好来选择使用哪种API。

总之，Vue 3提供了两种不同的API用于编写组件，选择哪种API取决于项目需求和个人偏好。对于较大型的组件和复杂的业务逻辑，建议使用Composition API来编写组件，而对于小型的组件或者对Vue 2.x较为熟悉的开发者，可以继续使用Options API来编写组件。

6.2.👉我👉: 都Vue3了就用Composition Api 吧(尽量别文艺复兴)

Composition Api的优点:

谢谢 [大帅老猿](#) 老师做的动图，Composition Api VS Option Api 的优缺点十分明了展示在了动画上！

1. Option Api 代码量少还好，代码量多容易导致高耦合！

option api 将功能分散在data、methods中如果功能模块太多,就太散

```
export default {
```

```
}
```

@稀土掘金技术社区

Options API

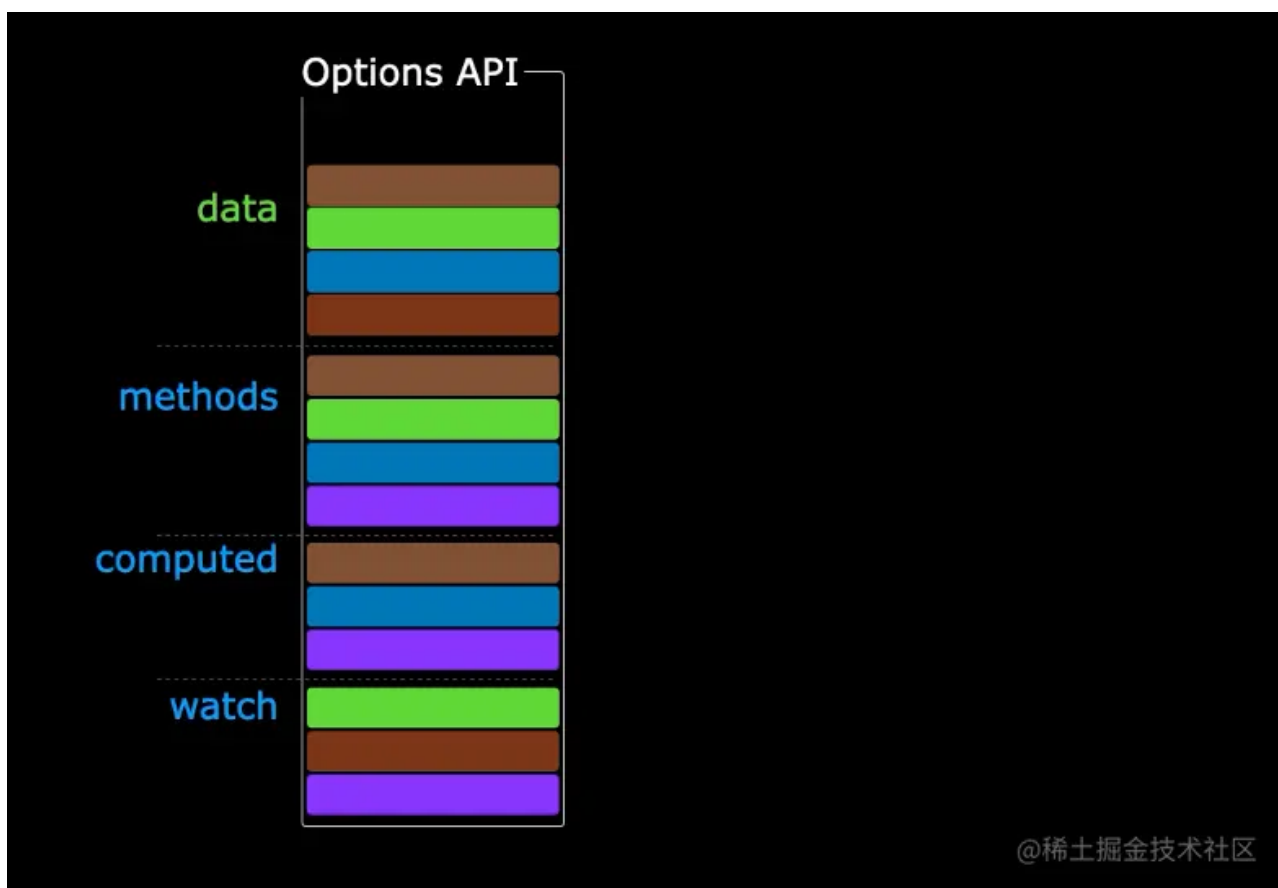
```
export default {  
  data() {  
    return {  
      功能 A  
      功能 B  
    };  
  },  
  methods: {  
    功能 A  
    功能 B  
  },  
  computed: {  
    功能 A  
  },  
  watch: {  
    功能 B  
  }  
}
```

@稀土掘金技术社区

说明：上面是 Vue2 Option Api 的写法，一个组件下含有 data 、methos、computed、watch ， 同一个功能需要分开写在这些函数上，如果代码量少，那看起来似乎十分明了清晰。一旦代码量大功能复杂，各个功能分开写，维护的时候 data 、methos、computed、watch 都需要来回切，反而显得**过于分散，又高度耦合**。

2. Composition Api 解耦 Option Api 实现低耦合高内聚

Composition Api 按模块划分代码块,功能更明确



说明：如果是 **Composition Api** 在功能复杂、代码量巨大的组件下，我们配合**自定义 Hook**，将代码**按功能分块写**，**变量和方法在一起定义和调用**，比如A功能下集成了响应式变量和方法，我们后期维护只需要改动A功能模块下的代码，不会像Vue2在 **Option Api** 需要同时关注逻辑分散的methos和data。

能使用 **Composition Api** 就尽量用,一些情况下迫不得已可以使用 **option Api**：

虽然能在Vue3中继续写 **Option Api** 但是2种写法是十分割裂的

- 如果项目是vue2改造过来的,可以用 **option Api** 过度一下
- 一些Vue2已经封装好的组件,开箱即用的,如果没有时间改造,可以直接继续使用

7. Vue3中组件通讯的方式

7.1 🤖GPT🤖: (答案有点问题混淆了vue2的内容)

父组件向子组件传递数据

0. props 子组件通过 props 属性从父组件接收数据。

1. *emit*事件子组件通过emit 发射事件向父组件发送消息。
2. provide / inject 父组件通过 provide 提供数据,子组件通过 inject 注入数据。
3. ref 父组件通过 ref 获取子组件实例,然后调用子组件的方法。
4. Slots 在父组件中使用Slots向子组件传递内容,子组件可以通过\$slots来获取这些内容,并将它们渲染到自己的视图中。

子组件向父组件传递数据

0. *emit*事件使用emit 发射事件,父组件通过 v-on 监听该事件。
1. provide/inject 子组件通过 inject 注入父组件提供的数据,然后触发 Action 修改该数据,父组件的数据也就变了。
2. Ref 父组件通过 ref 获取子组件的实例,子组件调用 ref 内容,父组件监听其变化。
3. Vuex 使用 Vuex 管理共享状态,子组件通过 mutation 或 action 修改状态,父组件监听状态变化。
4. Event Bus 利用 Vue 的实例作为 EventBus ,子组件通过 Bus.\$emit 触发事件,父组件监听该事件的变化。

兄弟组件之间的数据传递

0. EventBus 可以使用EventBus来创建一个全局的事件总线,从而在兄弟组件之间进行数据传递。通过在兄弟组件中使用*emit*和on来触发和监听事件,从而实现数据的传递。
1. Provide 和 Inject Vue 3中的Provide和Inject API也可以用于兄弟组件之间的数据传递,可以通过在父组件中使用Provide来提供数据,然后在兄弟组件中使用Inject来获取这些数据,从而实现兄弟组件之间的数据传递。
2. Vuex Vuex是Vue 3中的状态管理库,可以用于管理全局的数据状态,从而实现兄弟组件之间的数据传递。

7.2.👉我👉:

方式	Vue2	Vue3
父传子	props	props
父传子	provide/inject	provide/inject
父传子	\$attrs	attrs
父组件访问子组件	\$children	无
父组件访问子组件	\$ref	expose(defineExpose)&ref
子传父	\$emit	emits(defineEmits)
子传父	\$listeners	无(合并到 attrs方式)
子组件访问父组件	\$parent	无
兄弟传值	EventBus	mitt(需要引入插件实现)

父组件向子组件传递数据:(父传子)

1. props:

逐级透传

- 父组件



javascript 复制代码

```
<template>
  <ChildComponent :msg="message" />
</template>
<script setup>
  const message = 'hello'
</script>
```

- 子组件:



javascript 复制代码

```
<template>
  <p>{{ msg }}</p>
</template>

<script setup>
  defineProps(['msg'])
</script>
```

2. provide/inject

跨级共享

- provide(提供): 父组件中注册,传递给后代组件的数据对象
- inject(注入):后代组件中接收父组件提供的数据对象

- 父组件 Root.vue



javascript 复制代码

```
<template>
  <div>
    <Footer/>
  </div>
</template>
<script setup>
  import { ref,provide } from 'vue'
  import Footer from './Footer.vue' //导入子组件Footer
  //父组件中注册给共享后代组件的信息
  provide(/* 注入名 */ 'mgs', /* 值 */ '父组件信息!')
</script>
```

- 子组件 Footer.vue



javascript 复制代码

```
<template>
  <DeepChild/>
</template>
<script setup>
  import DeepChild from './DeepChild.vue' //导入子组件Footer
  //在子组件Footer中不需要接收父组件Root共享的数据,在后代组件中,直接接收,实现跨级传递
</script>
```

- 子孙组件DeepChild.vue

不需要父组件Footer传递,就能跨级获取祖先组件Root传递的数据

javascript 复制代码

```
<template>
  <div>
    {{ msg }}
  </div>
</template>
<script setup>
  import { inject } from 'vue'
  const msg = inject('msg') //跨级接收祖先组件的共享的信息
  console.log(msg) //父组件信息
</script>
```

父组件向子组件传递信息,如果是**多层组件嵌套(父>子>孙>孙孙.....)**,**props逐级透传**十分麻烦,props可用但不优雅,更推荐 **provide/inject依赖注入** 的方式;

在**** provide/inject依赖注入 ****中父组件向后代组件通讯,如果说props是传递,逐级透传的,那么依赖注入更准确来说是在父组件中与后代组件共享数据,可实现跨级共享;

3. 透传 Attributes(非props和非emit)

“透传 attribute”指的是传递给一个组件，却没有被该组件声明为 **props** 或 **emits** 的 attribute 或者 **v-on** 事件监听器。最常见的例子就是 **class**、**style** 和 **id**

作用: 在父组件标签上声明的**参数/事件监听**,会透传到子组件中

3.1 Attributes透传参数

- 父组件

html 复制代码

```
<!-- 透传参数class到子组件中 -->
<MyButton class="large" />
```

- 子组件 **MyButton.vue**


```
<button>click me</button>
```

- 最终渲染的html标签

```
<button class="btn large">click me</button>
```

3.2 Attributes透传事件监听(有点像冒泡事件)

- 父组件

```
<!-- 透传参数class到子组件中 -->
<MyButton @click="onClick1" />

<script setup>
  import MyButton from './MyButton.vue'
  const onClick = ()=>{
    console.log("透传事件监听,从父组件触发")
  }
</script>
```

- 子组件 `MyButton.vue`

```
<button @click ="onClick2">click me</button>
```

- 当点击子组件的按钮时:

父组件的 `onClick1` 和子组件的 `onClick2` 都触发

3.3 useAttrs 像defineProps获取透传Attributes

- 父组件

```
<!-- 透传参数ms到子组件中 -->
<Child msg="父组件中传递数据" />

<script setup>
  import Child from './child.vue'
</script>
```

- 子组件child.vue



javascript 复制代码

```
<script setup>
import { useAttrs } from 'vue'
//useAttrs像defineProps获取透传Attributes
const attrs = useAttrs()
console.log(attrs.msg) //父组件中传递数据
</script>
```

4. slot 插槽

父组件向子组件指定位置插入 **html** 内容渲染

4.1 默认插槽(传递html/组件)

- 父组件



html 复制代码

```
<template>
  <Child>
    <div>插入的html,将会在子组件中指定slot的位置渲染出来</div>
  </Child>
</template>
```

- 子组件



html 复制代码

```
<template>
  <slot></slot>
</template>
```

4.2 具名插槽(父->子)

当需要渲染不同的内容时,默认插槽显然不够用,需要按插槽的 `name` 名进行区别渲染

- 父组件

`#XX` == `v-slot:XX` 这2个写法都是插槽name名在父组件的写法



html 复制代码

```
<子>
  <template #XX1> 插槽1 </template>
  <template v-slot:XX2>插槽2</template>
</子>
```

- 子组件



html 复制代码

```
<template>
  <slot name="XX1"></slot>
  <slot name="XX2"></slot>
</template>
```

4.3 作用域插槽 (子>父)

作用域插槽分为: 默认作用域插槽和 具名作用域插槽

elementUI中table组件插入按钮就是使用了默认作用域插槽

4.3.1. 默认作用域插槽: `v-slot:default` = `#default` = `v-slot`

- 父组件



html 复制代码

```
<子>
  <template #default="slotProps">
    {{slotProps.XX}}
  </template>
```

```
</template>
</子>
```

- 子组件



ruby 复制代码

```
<slot :XX="子组件数据"></slot>
```

4.3.2. 具名作用域插槽

- 父组件



html 复制代码

```
<子>
  <template #slotName="slotProps">
    {{slotProps.XX}}
  </template>
</子>
```

- 子组件



html 复制代码

```
<slot name="slotName" :XX="子组件数据"></slot>
```

Element-Plus UI组件框架中table组件

table组件中就用到了作用域插槽



html 复制代码

```
<el-table :data="tableData" style="width: 100%" max-height="250">
  <el-table-column fixed prop="date" label="Date" width="150" />
  <el-table-column fixed="right" label="Operations" width="120">
    <template #default="scope">
      {{scope.row.date}}
    </template>
  </el-table-column>
</el-table>
```

```
</el-table-column>
</el-table>
```

子组件向父组件传递数据:(子传父)

1.组件事件emit

父组件中v-on(简写@)监听

子组件中\$emit触发

- 父组件

父组件中v-on(简写@)监听

html 复制代码

```
<template>
  <child @some-event="callback" />
</template>
<script setup>
import child from './child.vue'
const callback = (target) => {
  console.log('父组件-callback ')
  console.log(target) //子组件传递的数据
}
</script>
```

- 子组件

子组件中\$emit触发

html 复制代码

```
<template>
  <!-->在template中使用$emit触发,不需要defineEmits声明</-->
  <button @click="$emit('someEvent', '子组件传递的数据')">click me</button>
  <!-->触发方法中的emit,需要defineEmits声明</-->
  <button @click="buttonClick()">click me</button>
```

```
</template>

<script setup>
//setup语法糖中显示声明emit
const emit = defineEmits(['someEvent'])
function buttonClick() {
  //触发emit
  emit('someEvent', '子组件传递的数据')
}
</script>
```

2 defineExpose/ ref

子组件中通过 `defineExpose` 向外暴露数据或方法

父组件中通过`ref`获取子组件暴露的数据或调用子组件的方法

- 父组件

在父组件中需要声明子组件的 `ref` , 如: `const childRef = ref()`

html 复制代码

```
<template>
  <child ref="childRef" />
</template>
<script setup>
import { ref, onMounted } from 'vue'
// 引入子组件
import child from './child.vue'

const childRef = ref()
onMounted(() => {
  console.log(childRef.value.data1) // 子组件数据
  childRef.value.fn() // 子组件中的方法
})
</script>
```

- 子组件

```
<script setup>
import { ref } from 'vue'
const data1 = ref('子组件数据')
const fn = () => {
  console.log('子组件中的方法')
}
//通过defineExpose向外暴露数据或方法
defineExpose({
  data1,
  fn
})
</script>
```

跨组件通讯-全局状态共享(状态管理库): Vuex /Pinia

在Vue3已经逐渐用Pinia这个菠萝替代Vuex了

Pinia,官方文档描述:符合直觉的 **Vue.js 状态管理库** hook的写法!

其实Pinia的官方文档就写得很清楚了:[🐼🐼🐼🐼🐼🐼🐼🐼🐼🐼🐼🐼🐼🐼🐼🐼为什么你应该使用 Pinia?](#)

8 . Vue2和Vue3的区别?

8.1 🤖 GPT 🤖:(概括得很好!)

8.1.1. setup()

Vue3 通过新的 `setup()` 选项来取代 Vue2 中的 `created/mounted` 等函数。

8.1.2. 组合式API : composition API

Vue3 引入了许多新的 composition API,比如:

- readonly()
- ref()
- reactive()
- computed()
- watchEffect()
- onMounted() 等等

8.1.3.生命周期钩子

Vue3 大幅简化了生命周期钩子,只保留了最基本的:

- setup()
- onMounted()
- onUpdated()
- onUnmounted()

8.1.4.响应性原理

Vue3 使用 Proxy API 来实现响应性,Vue2 使用了 Observer模式。Proxy API 更高效、灵活。

8.1.5. 性能提升

Vue3 通过改变响应性原理和优化 Virtual DOM 算法,性能有着明显提升。

8.1.6.Fragment

Vue3 默认支持 `<template></template>` 来作为根元素,而不是 `<div>` 。

8.1.7.Teleport

Vue3 引入了 `<Teleport>` 组件,允许你瞬移 DOM 节点变更渲染位置。

8.1.8.Vue Router

Vue Router 在 Vue3 中也做了不少改进,但最大的区别在 API 重构。

8.1.9.Keep Alive

<keep-alive> 增强了 includes、excludes属性和 activated 和 deactivated 生命周期钩子

8.2.👉我👉:

GPT的答案已经很好了

大致对比

区别项	Option API	Composition API
定义方式	使用 options 对象配置组件(data 、 methods ...)	使用 setup() 函数配置组件
数据来源	数据定义在 data 函数返回对象中	数据定义在 setup() 函数中使用ref/reactive 声明
方法	定义在 methods 对象中	定义在 setup() 函数中
侦听器	使用 watch 指定侦听	使用 watch 和 watchEffect 实现侦听
计算属性	使用 computed 定义	使用 computed 函数定义
提取数据	使用 provide/inject	使用 provide/inject
模块与组件间通信	使用 \$on / \$emit	使用 emit() 绑定到父组件
异步操作	在 created 生命周期中调用	在 setup() 中直接调用
代码复用性	通过混入(mixins)和继承实现代码复用	自定义 Hook
访问状态和方法	使用 this 关键字	setup 语法糖中直接访问状态和方法

生命周期Hook对比:

Option API	Composition API
beforeCreate	setup()
created	setup()
beforeMount	onBeforeMount
mounted	onMounted
beforeUpdate	onBeforeUpdate
updated	onUpdated
beforeDestroy	onBeforeUnmount
destroyed	onUnmounted (这个区别较大)
errorCaptured	onErrorCaptured

路由对比

Option API	Composition API
this.\$router	useRouter()
this.\$route	useRoute()

1. 数据驱动能力更强，重构响应式，基于 **proxy** 的数据驱动能够弥补原来 **defineProperty** 的不足,

使用Proxy的优势：

1. Proxy代理整个对象，Object.defineProperty只代理对象上的某个属性(更直观)
2. Proxy可直接监听数组类型的数据变化
3. 可拦截apply、ownKeys、has等方法，而Object.defineProperty不行

4. Proxy直接实现对象属性的新增/删除,依旧能监听

vue2中对象新增属性,需要使用\$set()保持响应;

vue3中如果是使用TS也不能向对象新增属性,TS会报错(对象不存在该属性)

Proxy和Object.defineProperty对比:

javascript 复制代码

```
//user的对象包含name和age属性
const user = {
  name: 'John',
  age: 30
}
//proxy代理整个对象,它可以捕获对user对象的访问和修改
const userProxy = new Proxy(user, {
  get(target, key) {
    console.log(`Getting ${key} property`);
    return target[key];
  },
  set(target, key, value) {
    console.log(`Setting ${key} property to ${value}`);
    target[key] = value;
    return true;
  }
});
//Object.defineProperty 对象的属性需要一个个代理,十分麻烦,更加冗长和复杂
Object.defineProperty(user, 'name', {
  get() {
    console.log(`Getting name property`);
    return user.name;
  },
  set(value) {
    console.log(`Setting name property to ${value}`);
    user.name = value;
  }
});

Object.defineProperty(user, 'age', {
  get() {
    console.log(`Getting age property`);
    return user.age;
  },
  set(value) {
    console.log(`Setting age property to ${value}`);
    user.age = value;
  }
});
```

```
}  
});
```

2. 新增 **Composition API** , 更好的逻辑复用和代码组织

按功能划分代码块,更直观

3. 重构 Virtual DOM



markdown 复制代码

1. 模板编译时的优化, 将一些静态节点编译成常量
2. slot优化, 将slot编译为lazy函数, 将slot的渲染的决定权交给子组件
3. 模板中内联事件的提取并重用 (原本每次渲染都重新生成内联函数)

4. 代码结构调整, 更有利于Tree shaking, 使得体积更小(速度快体积小)

5. Vue3全面拥抱 typescript, 2.x 版本无论用 class component 还是 配置 都不能很好的支持 ts.

6. Vue3已经不兼容IE11

7. Vue3支持自定义Hook替代Mixin

😁😁😁如果觉得写得还行, 不吝啬点个赞再走!笔芯!本文会持续更新~~~😁😁😁😁

标签: **Vue.js** **前端框架** **JavaScript**