为啥面试官总喜欢问computed是咋实现的?

从computed的特性出发

`computed` 最耀眼的几个特性是啥?

1. **依赖追踪**

- javascript
  复制代码
  import { reactive, computed } from 'vue'const state = reactive({ a: 1, b: 2, c: 3,})const sum = computed(() => { return state.astate.b})

我们定义了一个响应式数据 `state` 和一个计算属性 `sum` , Vue会自动追踪 `sum` 依赖的数据 `state.a` 和 `state.b` ，并建立相应的依赖关系。

也就是只有 `state.a` 和 `state.b` 发生变化的时候， `sum` 才会重新计算而 `state.c` 任由它怎么变， `sum` 都将丝毫不受影响。

2. **缓存**

还是上面的例子，如果 `state.a` 和 `state.b` 打死都不再改变值了，那么我们读取 `sum` 的时候，它将会返回上一次计算的结果，而不是重新计算。

3. **懒计算**

这个特性比较容易被忽略，简单地说只有计算属性真正被使用（读取）的时候才会进行计算，否则咱就仅仅是定义了一个变量而已。

- javascript
  复制代码
  import { reactive, computed } from 'vue'const state = reactive({ a: 1, b: 2, c: 3})const sum = computed(() => { console.log('执行计算') return state.astate.b})setTimeout(() => { // 没有读取sum.value之前，sum不会进行计算 console.log('1-sum', sum.value) // 我们改变了a的值，但是sum并不会立刻进行计算 state.a = 4 setTimeout(() => { // 而是要等到再次读取的时候才会触发重新计算 console.log('2-sum', sum.value) }, 1000)}, 1000)

挨个实现computed特性

4. 懒计算

我们依旧围绕 `effect` 函数来搞事情，到目前为止， `effect` 注册的回调都是立刻执行。

- javascript
  复制代码
  const state = reactive({ a: 1, b: 2, c: 3})// 有没有很像计算属性的感觉const sum = effect(() => { console.log('执行计算') // 立刻被打印 const value = state.astate.b return value})console.log(sum) // undefined

想要实现 `computed` 的懒执行，咱们可以参考上篇文章Vue3：原来你是这样的“异步更新”的思路，添加一个额外的参数 `lazy` 。

它要实现的功能是：如果传递了 `lazy` 为 `true` ，副作用函数将不会立即执行，而是将执行的时机交还给用户，由用户决定啥时候执行。

当然啦！回调的结果我们也应该一并返回（例如上面的value值）

你能想象，我们仅仅需要改造几行代码就能离 `computed` 近了一大步。

```
1 javascript
2 复制代码
3 const effect = function (fn, options = {}) {  const effectFn = () => {    // ... 省略    // 新增res存储fn执行的结果    const res = fn()    // ... 省略    // 新增返回结果    return res  }  // ... 省略  // 新增，只有lazy不为true时才会立即执行  if (!options.lazy) {    effectFn()  }  // 新增，返回副作用函数让用户执行  return effectFn}
```

测试一波

- javascript
  复制代码
  const state = reactive({ a: 1, b: 2, c: 3,});// 有没有很像计算属性的感觉const sum = effect(() => { console.log("执行计算"); // 调用sum函数后被打印 const value = state.astate.b; return value;}, { lazy: true});// 不执行sum函数，effect注册的回调将不会执行console.log(sum()); // 3

5. 依赖追踪

咱们初步实现了**懒执行**的特性，为了更像 `computed` 一点，我们需要封装一个函数。

```
1 javascript
2 复制代码
3 function computed (getter) {  const effectFn = effect(getter, {    lazy: true,  })  const obj = {    get value () {      return effectFn()    }  }  return obj}
```

这就有点那么味道啦！

测试一波

可以看到 `computed` 只会依赖 `state.a` 和 `state.b` ，而不会依赖 `state.c` ，这得益于我们前面几篇文章实现的响应式系统，所以到了计算属性这里，我们不用改动任何代码，天然就支持。

不过还是有点小问题，我们读取了两次 `sum.value` ，sum却被执行了两次，这和 `computed` 缓存的特性就不符了。

别急，马上就要实现了这个最重要的特性了。

- javascript
  复制代码
  const state = reactive({ a: 1, b: 2, c: 3})const sum = computed(() => { console.log('执行计算') return state.astate.b})console.log(sum.value)console.log(sum.value)

6. **缓存**

回顾一下 `computed` 的缓存特性:

1. 只有当其依赖的东西发生变化了才需要重新计算
2. 否则就返回上一次执行的结果。

为了缓存上一次计算的结果，咱们需要定义一个value变量，现在的关键是**怎么才能知道其依赖的数据发生变化了呢?**

```
1 javascript
2 复制代码
3 function computed (getter) {  const effectFn = effect(getter, {    lazy: true,
   })  let value  let dirty = true  const obj = {    get value () {      // 2. 只
  有数据发生变化了才去重新计算      if (dirty) {        value = effectFn()
  dirty = false      }      return value    }  }  return obj}
```

测试一波

- javascript
  复制代码
  const state = reactive({ a: 1, b: 2, c: 3})const sum = computed(() => { console.log('执行计算') return state.astate.b})console.log(sum.value) // 3console.log(sum.value) // 3state.a = 4console.log(sum.value) // 3 答案是错误的

寄上任务调度

不得不说，任务调度实在太强大了，不仅仅可以实现数组的异步批量更新、在 `computed` 和 `watch` 中也是必不可少的。

```
1 javascript
2 复制代码
3 function computed (getter) {  const effectFn = effect(getter, {    lazy: true,
    // 数据发生变化后，不执行注册的回调，而是执行scheduler    scheduler () {      //
  数据发生了变化后，则重新设置为dirty，那么下次就会重新计算      dirty = true    }  })
  let value  let dirty = true  const obj = {    get value () {      // 2. 只有数据
  发生变化了才去重新计算      if (dirty) {        value = effectFn()        dirty =
  false      }      return value    }  }  return obj}
```

测试一波

- javascript
  复制代码
  const state = reactive({ a: 1, b: 2, c: 3})const sum = computed(() => { console.log('执行计算')
  return state.astate.b})console.log(sum.value) // 3console.log(sum.value) // 3state.a =
  4console.log(sum.value) // 3 答案是错误的

完美！！！这下面试官再也难不倒我了！！！