

Composition API (—)

王红元 coderwhy

- 目前我们是使用组件化的方式在开发整个Vue的应用程序，但是**组件和组件之间有时候会存在相同的代码逻辑**，我们希望对**相同的代码逻辑进行抽取**。
- 在Vue2和Vue3中都支持的一种方式就是**使用Mixin来完成**：
 - Mixin提供了一种非常灵活的方式，来**分发Vue组件中的可复用功能**；
 - 一个Mixin对象可以包含**任何组件选项**；
 - 当组件使用Mixin对象时，所有**Mixin对象的选项将被 混合 进入该组件本身的选项中**；

Mixin的基本使用

V Home.vue U X

src > 14_Mixin混入 > pages > V Home.vue > {} "Home.vue"

```
1 <template>
2   <div>
3     <button @click="foo">foo点击</button>
4   </div>
5 </template>
6
7 <script>
8   import sayHelloMixin from '../mixins/sayHello';
9
10  export default {
11    mixins: [sayHelloMixin]
12  }
13 </script>
14
15 <style scoped>
16
17 </style>
```

JS sayHello.js U X

src > 14_Mixin混入 > mixins > JS sayHello.js > [E] default

```
1 const sayHelloMixin = {
2   created() {
3     this.sayHello();
4   },
5   methods: {
6     sayHello() {
7       console.log("Hello Page Component");
8     }
9   }
10 }
11
12 export default sayHelloMixin;
```

Mixin的合并规则

■ 如果Mixin对象中的选项和组件对象中的选项发生了冲突，那么Vue会如何操作呢？

□ 这里分成不同的情况来进行处理；

■ 情况一：如果是data函数的返回值对象

□ 返回值对象默认情况下会进行合并；

□ 如果data返回值对象的属性发生了冲突，那么会保留组件自身的数据；

■ 情况二：如何生命周期钩子函数

□ 生命周期的钩子函数会被合并到数组中，都会被调用；

■ 情况三：值为对象的选项，例如 methods、components 和 directives，将被合并为同一个对象。

□ 比如都有methods选项，并且都定义了方法，那么它们都会生效；

□ 但是如果对象的key相同，那么会取组件对象的键值对；

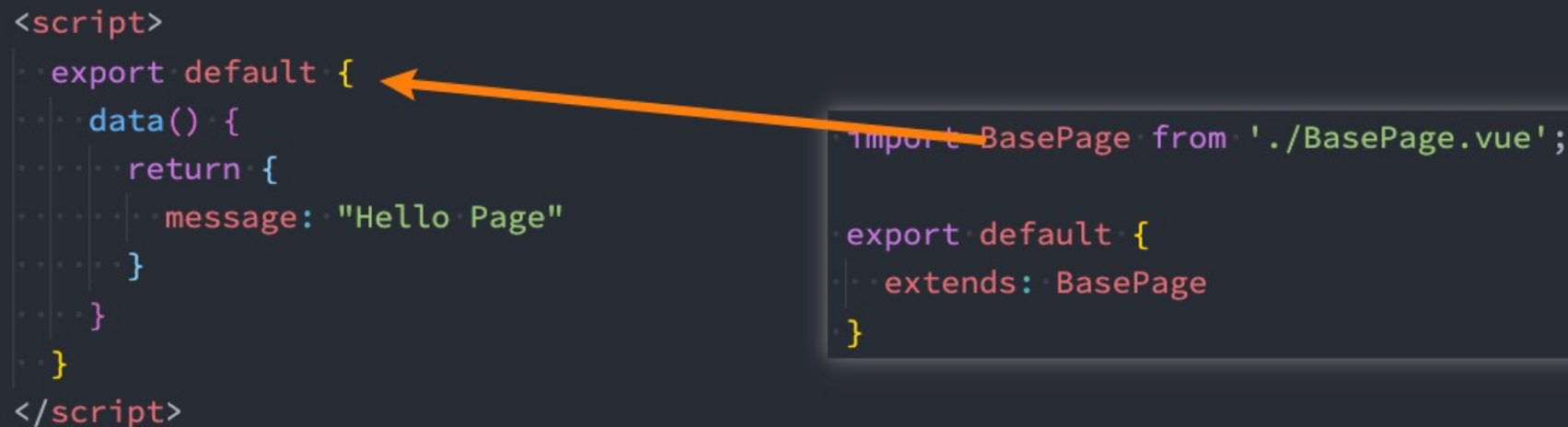
全局混入Mixin

- 如果组件中的某些选项，是所有的组件都需要拥有的，那么这个时候我们可以使用**全局的mixin**：
 - 全局的Mixin可以使用 **应用app的方法 mixin** 来完成注册；
 - 一旦注册，那么**全局混入的选项将会影响每一个组件**；

```
const app = createApp(App);
app.mixin({
  created() {
    console.log("global mixin created");
  }
})
app.mount("#app");
```

■ 另外一个类似于Mixin的方式是**通过extends属性**：

□ 允许声明扩展另外一个组件，**类似于Mixins**；



```
<script>
  export default {
    data() {
      return {
        message: "Hello Page"
      }
    }
  }
</script>
```

```
import BasePage from './BasePage.vue';

export default {
  extends: BasePage
}
```

■ 在开发中**extends用的非常少**，在Vue2中比较**推荐大家使用Mixin**，而在Vue3中**推荐使用Composition API**。



Options API的弊端

■ 在Vue2中，我们编写组件的方式是Options API：

□ Options API的一大特点就是在对应的属性中编写对应的功能模块；

□ 比如data定义数据、methods中定义方法、computed中定义计算属性、watch中监听属性改变，也包括生命周期钩子；

■ 但是这种代码有一个很大的弊端：

□ 当我们实现某一个功能时，这个功能对应的代码逻辑会被拆分到各个属性中；

□ 当我们组件变得更大、更复杂时，逻辑关注点的列表就会增长，那么同一个功能的逻辑就会被拆分的很分散；

□ 尤其对于那些一开始没有编写这些组件的人来说，这个组件的代码是难以阅读和理解的（阅读组件的其他人）；

■ 下面我们来看一个非常大的组件，其中的逻辑功能按照颜色进行了划分：

□ 这种碎片化的代码使用理解和维护这个复杂的组件变得异常困难，并且隐藏了潜在的逻辑问题；

□ 并且当我们处理单个逻辑关注点时，需要不断的跳到相应的代码块中；

大组件的逻辑分散

```

export default {
  data () {
    return {
      loading: false,
      error: false,
      editingPath: false,
      editingPath: '',
      folderCurrent: 0,
      foldersFavorite: [],
      showNewFolder: false,
      newFolderName: ''
    }
  },

  apollo: {
    folderCurrent: {
      query: FOLDER_CURRENT,
      fetchPolicy: 'network-only',
      loadingKey: 'loading',
      async result () {
        await this.$store.dispatch('setFolderCurrent', 0)
      }
    },
    foldersFavorite: FOLDERS_FAVORITE
  },

  computed: {
    newFolderName () {
      return this.$store.state.newFolderName
    }
  },

  watch: {
    showNewFolder: {
      if (value) {
        this.$store.dispatch('setFolderCurrent', 0)
      } else {
        this.$store.dispatch('setFolderCurrent', 0)
      }
    }
  },

  beforeRouteLeave (to, from, next) {
    if (this.$route.meta !== 'meta.newProject') {
      this.$store.dispatch('setFolderCurrent', 0)
    }
    next()
  },

  methods: {
    async openFolder (path) {
      this.loadingPath = false
      this.error = null
      this.loading = true
      try {
        await this.$store.dispatch('setFolderCurrent', path)
        mutation: FOLDER_OPEN,
        variables: {
          path
        },
        update: (store, { data: { folderOpen } }) => {
          store.writeQuery({ query: FOLDER_CURRENT, data: { folderCurrent: folderOpen } })
        }
      } catch (err) {
        this.error = err
      }
      this.loading = false
    },

    async openParentFolder (folder) {
      this.loadingPath = false
      this.error = null
      this.loading = true
      try {
        await this.$store.dispatch('setFolderCurrent', folder)
        mutation: FOLDER_OPEN_PARENT,
        update: (store, { data: { folderOpenParent } }) => {
          store.writeQuery({ query: FOLDER_CURRENT, data: { folderCurrent: folderOpenParent } })
        }
      } catch (err) {
        this.error = err
      }
      this.loading = false
    }
  }
}

```

```

100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000

```

- 如果我们能将同一个逻辑关注点相关的代码收集在一起会更好。
- 这就是Composition API想要做的事情，以及可以帮助我们完成的事情。
- 也有人把Vue Composition API简称为VCA。

认识Composition API

■ 那么既然知道Composition API想要帮助我们做什么事情，接下来看一下**到底是怎么做呢？**

□ 为了开始使用Composition API，我们需要有一个可以实际使用它（**编写代码**）的地方；

□ 在Vue组件中，这个位置就是 **setup 函数**；

■ **setup**其实就是组件的另外一个选项：

□ 只不过这个选项强大到我们可以**用它来替代之前所编写的大部分其他选项**；

□ 比如**methods、computed、watch、data、生命周期**等等；

■ 接下来我们一起学习这个函数的使用：

□ 函数的参数

□ 函数的返回值



setup函数的参数

- 我们先来研究一个setup函数的参数，它主要有两个参数：
 - 第一个参数：**props**
 - 第二个参数：**context**
- props非常好理解，它其实就是父组件传递过来的属性会被放到props对象中，我们在setup中如果需要使用，那么就可以直接通过props参数获取：
 - 对于定义props的类型，我们还是和之前的规则是一样的，在props选项中定义；
 - 并且在template中依然是可以正常去使用props中的属性，比如message；
 - 如果我们想在setup函数中使用props，那么不可以通过 this 去获取（后面我会讲到为什么）；
 - 因为props有直接作为参数传递到setup函数中，所以我们可以直接通过参数来使用即可；
- 另外一个参数是context，我们也称之为是一个SetupContext，它里面包含三个属性：
 - **attrs**：所有的非prop的attribute；
 - **slots**：父组件传递过来的插槽（这个在以渲染函数返回时会有作用，后面会讲到）；
 - **emit**：当我们组件内部需要发出事件时会用到emit（因为我们不能访问this，所以不可以通过 this.\$emit发出事件）；

setup函数的返回值

■ setup既然是一个函数，那么它也可以有**返回值**，它的返回值用来做什么呢？

- setup的返回值可以在**模板template**中被使用；
- 也就是说我们可以**通过setup的返回值来替代data选项**；

■ 甚至是我们可以**返回一个执行函数来代替在methods中定义的方法**：

```
const name = "coderwhy";
let counter = 100;
const increment = () => {
  counter++;
}
const decrement = () => {
  counter--;
}
```

```
return {
  name,
  counter,
  increment,
  decrement
}
```

■ 但是，如果我们将 counter 在 increment 或者 decrement 进行操作时，**是否可以实现界面的响应式呢？**

- 答案是**不可以**；
- 这是因为对于一个**定义的变量**来说，默认情况下，**Vue并不会跟踪它的变化**，来引起界面的响应式操作；

setup不可以使用this

■ 官方关于this有这样一段描述（这段描述是我给官方提交了PR之后的一段描述）：

- 表达的含义是this并没有指向当前组件实例；
- 并且在setup被调用之前，data、computed、methods等都没有被解析；
- 所以无法在setup中获取this；



WARNING

在 `setup` 中你应该避免使用 `this`，因为它不会找到组件实例。`setup` 的调用发生在 `data` property、`computed` property 或 `methods` 被解析之前，所以它们无法在 `setup` 中被获取。

■ 其实在之前的这段描述是和源码有出入的（我向官方提交了PR，做出了描述的修改）：

- 之前的描述大概含义是不可以使用this是因为组件实例还没有被创建出来；
- 后来我的PR也有被合并到官方文档中；

之前关于this的描述问题



coderwhy commented 24 days ago

Contributor ...

Description of Problem

According to the content of the vue3 source code, the component instance has been created when the setup is executed. But the description in the document is not created, so `this` cannot be used.

According to the original, `this` cannot be used because this is not bound to this during setup, and props, etc. cannot be used because options such as props, data, compute, etc. will be processed later.

```
// 1. 调用createComponentInstance创建组件的实例
const instance: ComponentInternalInstance = (initialVNode.component = createComponentInstance(
  initialVNode,
  parentComponent,
  parentSuspense
))

if (__DEV__ && instance.type.__hmrId) { ...
}

if (__DEV__) { ...
}

// inject renderer internals for keepAlive
if (isKeepAlive(initialVNode)) { ...
}

// resolve props and slots for setup context
if (__DEV__) {
  startMeasure(instance, `init`)
}

// 2. setup组件实例, 作用是对组件的props/slots/data等进行初始化处理
// 并且内部有对vue2的options api进行兼容
setupComponent(instance)

if (__DEV__) {
```

我是如何发现官方文档的错误呢？

- 在阅读源码的过程中，代码是按照如下顺序执行的：
 - 调用 `createComponentInstance` 创建组件实例；
 - 调用 `setupComponent` 初始化component内部的操作；
 - 调用 `setupStatefulComponent` 初始化有状态的组件；
 - 在 `setupStatefulComponent` 取出了 `setup` 函数；
 - 通过 `callWithErrorHandling` 的函数执行 `setup`；
- 从上面的代码我们可以看出，**组件的instance肯定是在执行 `setup` 函数之前就创建出来的。**

```
export function callWithErrorHandling(  
  fn: Function,  
  instance: ComponentInternalInstance | null,  
  type: ErrorTypes,  
  args?: unknown[]  
) {  
  let res  
  try {  
    res = args ? fn(...args) : fn()  
  } catch (err) {  
    handleError(err, instance, type)  
  }  
  return res  
}
```



Reactive API

- 如果想为在setup中定义的数据提供响应式的特性，那么我们可以使用reactive的函数：

```
const state = reactive({  
  name: "coderwhy",  
  counter: 100  
})
```

- 那么这是为什么呢？为什么就可以变成响应式的呢？

- 这是因为当我们使用reactive函数处理我们的数据之后，数据再次被使用时就会进行依赖收集；
- 当数据发生改变时，所有收集到的依赖都是进行对应的响应式操作（比如更新界面）；
- 事实上，我们编写的data选项，也是在内部交给了reactive函数将其编程响应式对象的；

■ reactive API对传入的类型是有限制的，它要求我们必须传入的是一个对象或者数组类型：

□ 如果我们传入一个基本数据类型（String、Number、Boolean）会报一个警告；

```
▶ value cannot be made reactive: Hello World
```

■ 这个时候Vue3给我们提供了另外一个API：ref API

□ ref 会返回一个可变的响应式对象，该对象作为一个响应式的引用维护着它内部的值，这就是ref名称的来源；

□ 它内部的值是在ref的 value 属性中被维护的；

```
const message = ref("Hello World");
```

■ 这里有两个注意事项：

□ 在模板中引入ref的值时，Vue会自动帮助我们进行解包操作，所以我们并不需要在模板中通过 ref.value 的方式来使用；

□ 但是在 setup 函数内部，它依然是一个 ref引用，所以对其进行操作时，我们依然需要使用 ref.value的方式；

Ref自动解包

- 模板中的解包是浅层的解包，如果我们的代码是下面的方式：
- 如果我们~~将ref~~放到一个reactive的属性当中，那么在模板中使用时，它会自动解包：

```
<template>
  <div>
    <h2>{{message}}</h2>
    <h2>{{info.message.value}}</h2>
    <button @click="changeMessage">changeMessage</button>
  </div>
</template>

<script>
  import { ref } from 'vue';

  export default {
    setup() {
      const message = ref("Hello World");
      const changeMessage = () => message.value = "你好啊，李银河";

      const info = {
        message
      }

      return {
        message,
        changeMessage,
        info
      }
    }
  }
}
```

```
<template>
  <div>
    <h2>{{message}}</h2>
    <h2>{{info.message}}</h2> 这里不需要.value
    <button @click="changeMessage">changeMessage</button>
  </div>
</template>

<script>
  import { ref, reactive } from 'vue';

  export default {
    setup() {
      const message = ref("Hello World");
      const changeMessage = () => message.value = "你

      const info = reactive({
        message
      })

      return {
        message,
        changeMessage,
        info
      }
    }
  }
}
```

认识readonly

- 我们通过**reactive**或者**ref**可以获取到一个响应式的对象，但是某些情况下，我们传入给其他地方（组件）的这个响应式对象希望在另外一个地方（组件）被使用，但是**不能被修改**，这个时候如何防止这种情况的出现呢？
 - Vue3为我们提供了**readonly**的方法；
 - **readonly**会返回原生对象的只读代理（也就是它依然是一个Proxy，这是一个**proxy**的**set**方法被劫持，并且不能对其进行修改）；
- 在开发中常见的**readonly**方法会传入三个类型的参数：
 - 类型一：**普通对象**；
 - 类型二：**reactive**返回的对象；
 - 类型三：**ref**的对象；

readonly的使用

■ 在readonly的使用过程中，有如下规则：

- readonly返回的对象都是不允许修改的；
- 但是经过readonly处理的原来的对象是允许被修改的；
 - ✓ 比如 `const info = readonly(obj)`，`info`对象是不允许被修改的；
 - ✓ 当`obj`被修改时，`readonly`返回的`info`对象也会被修改；
 - ✓ 但是我们不能去修改`readonly`返回的对象`info`；
- 其实本质上就是`readonly`返回的对象的`setter`方法被劫持了而已；

```
// readonly通常会传入三个类型的数据
// 1. 传入一个普通对象
const info = {
  name: "why",
  age: 18
}
const state1 = readonly(info)

console.log(state1);

// 2. 传入reactive对象
const state = reactive({
  name: "why",
  age: 18
})
const state2 = readonly(state);

// 3. 传入ref对象
const nameRef = ref("why");
const state3 = readonly(nameRef);
```

readonly的应用

■ 那么这个readonly有什么用呢？

- 在我们传递给其他组件数据时，往往希望其他组件使用我们传递的内容，但是不允许它们修改时，就可以使用readonly了；

```
05_readonly-案例.vue
3 <h2>{{info.name}}</h2>
4 <h2>{{info.age}}</h2>
5
6 <home :info="info"/>
7 </div>
8 </template>
9
10 <script>
11 import { reactive } from 'vue';
12
13 import Home from './pages/Home.vue';
14
15 export default {
16   components: {
17     Home
18   },
19   setup() {
20     const info = reactive({
21       name: "why",
22       age: 18
23     });
24
25     return {
26       info
27     }
28   }
29 }
30 </script>
```

```
Home.vue
<template>
  <div>
    <h2>Home: {{info.name}}</h2>
    <button @click="changeName">修改name</button>
  </div>
</template>

<script>
export default {
  props: {
    info: Object
  },
  setup(props) {
    const changeName = () => {
      props.info.name = "home";
    }

    return {
      changeName
    }
  }
}</script>
```

```
<home :info="readonlyInfo"/>
</div>
</template>

<script>
import { reactive, readonly } from 'vue';

import Home from './pages/Home.vue';

export default {
  components: {
    Home,
    About
  },
  setup() {
    const info = reactive({
      name: "why",
      age: 18
    });

    const readonlyInfo = readonly(info);

    return {
      info,
      readonlyInfo
    }
  }
}
```

Reactive判断的API

■ isProxy

- 检查对象是否是由 reactive 或 readonly创建的 proxy。

■ isReactive

- 检查对象是否是由 reactive创建的响应式代理：
- 如果该代理是 readonly 建的，但包裹了由 reactive 创建的另一个代理，它也会返回 true；

■ isReadonly

- 检查对象是否是由 readonly 创建的只读代理。

■ toRaw

- 返回 reactive 或 readonly 代理的原始对象（不建议保留对原始对象的持久引用。请谨慎使用）。

■ shallowReactive

- 创建一个响应式代理，它跟踪其自身 property 的响应性，但不执行嵌套对象的深层响应式转换（深层还是原生对象）。

■ shallowReadonly

- 创建一个 proxy，使其自身的 property 为只读，但不执行嵌套对象的深度只读转换（深层还是可读、可写的）。

- 如果我们使用ES6的解构语法，对reactive返回的对象进行解构获取值，那么之后无论是修改结构后的变量，还是修改reactive返回的state对象，**数据都不再是响应式**的：

```
const state = reactive({
  name: "why",
  age: 18
});

const { name, age } = state;
```

- 那么有没有办法让我们解构出来的属性是响应式的呢？

- Vue为我们提供了一个**toRefs**的函数，可以将reactive返回的对象中的属性都转成ref；

- 那么我们再次进行结构出来的 **name** 和 **age** 本身都是 **ref**的；

```
// 当我们这样做的时候，会返回两个ref对象，它们是响应式的
const { name, age } = toRefs(state);
```

- 这种做法相当于已经在**state.name**和**ref.value**之间建立了 **链接**，任何一个修改都会引起另外一个变化；

- 如果我们只希望转换一个reactive对象中的属性为ref, 那么可以使用toRef的方法：

```
// 如果我们只希望转换一个reactive对象中的属性为ref, 那么可以使用toRef的方法  
const name = toRef(state, 'name');  
const {age} = state;  
const changeName = () => state.name = "coderwhy";
```


■ unref

■ 如果我们想要**获取一个ref引用中的value**，那么也可以**通过unref方法**：

- 如果参数是一个 **ref**，则**返回内部值**，否则返回参数本身；
- 这是 `val = isRef(val) ? val.value : val` 的语法糖函数；

■ isRef

- 判断值**是否是一个ref对象**。

■ shallowRef

- 创建一个**浅层的ref对象**；

■ triggerRef

- **手动触发和 shallowRef 相关联的副作用**：

```
const info = shallowRef({name: "why"});  
  
// 下面的修改不是响应式的  
const changeInfo = () => {  
  info.value.name = "coderwhy"  
  // 手动触发  
  triggerRef(info);  
};
```


- 创建一个自定义的ref，并对其依赖项跟踪和更新触发进行显示控制：
 - 它需要一个工厂函数，该函数接受 track 和 trigger 函数作为参数；
 - 并且应该返回一个带有 get 和 set 的对象；
- 这里我们使用一个的案例：
 - 对双向绑定的属性进行debounce(节流)的操作；

customRef的案例

```
import { customRef } from 'vue';

export function useDebouncedRef(value, delay = 200) {
  let timeout;
  return customRef((track, trigger) => {
    return {
      get() {
        track();
        return value;
      },
      set(newValue) {
        clearTimeout(timeout);
        timeout = setTimeout(() => {
          value = newValue;
          trigger();
        }, delay);
      }
    }
  })
}
```

```
<template>
  <div>
    <input v-model="message">
    <h2>{{message}}</h2>
  </div>
</template>

<script>
  import { useDebouncedRef } from '../hooks/useDebounceRef';

  export default {
    setup() {
      const message = useDebouncedRef("Hello World");
      return {
        message
      }
    }
  }
</script>
```