# setup

- setup执行的时机
  - 在beforeCreate之前执行(一次), 此时组件对象还没有创建
  - setup在执行的时候，当前的组件还没有创建出来，组件实例对象this根本就不能用，this是undefined, 不能通过this来访问data/computed/methods/props
  - 其实所有的composition API相关回调函数中也都不可以
- setup的返回值
  - 一般都返回一个对象: 为模板提供数据, 内部的属性和方法是给html模版使用的
  - setup中的对象中的属性和data函数中的return对象的属性合并为组件对象的属性，都可以在html模版中使用
  - setup中的对象中的方法和methods对象中的方法会合并为组件对象的方法
  - 如果有重名, setup优先
  - 注意:
    - 一般不要混合使用：methods中可以访问setup提供的属性和方法, 但在setup方法中不能访问data和methods
    - setup不能是一个async函数：因为返回值不再是return的对象, 而是promise, 模板看不到return对象中的属性数据
- setup的参数
  - setup(props, context) / setup(props, {attrs, slots, emit})
  - props：包含props配置声明接收且传入了的所有属性的对象
  - context
    - attrs：包含没有在props配置中声明接收的属性的对象, 相当于 this.$attrs
    - slots：包含所有传入的插槽内容的对象, 相当于 this.$slots
    - emit：用来分发自定义事件的函数, 相当于 this.$emit

# ref与reactive

- ref用来处理基本类型数据，reactive用来处理对象(递归深度响应式)
- 如果用ref处理对象/数组, 内部会自动将对象/数组转换为reactive的代理对象，一个Proxy类型的对象
- ref内部：通过给value属性添加getter/setter来实现对数据的劫持
- reactive内部：通过使用Proxy来实现对对象内部所有数据的劫持, 并通过Reflect操作对象内部数据
- ref的数据操作：在js中要.value, 在模板中不需要(内部解析模板时会自动添加.value)

## ref

定义基本类型的响应式

```
setup(){
  // let count = 0 // 此时的数据并不是响应式的数据(响应式数据:数据变化,页面跟着渲染变化)
```

```
  // ref是一个函数,作用:定义一个响应式的数据,返回的是一个Ref对象,对象中有一个value属性,如果需要对数据进行操作,需要使用该Ref对象调用value属性的方式进行数据的操作
  // html模版中不需要使用.value属性的写法
  // count 的类型 Ref类型
  const count = ref(0)
  // 方法
  function updateCount(){
    // 报错的原因:count是一个Ref对象,对象是不能进行++的操作
    // count++
    count.value++
  }
  // 返回的是一个对象
  return {
    // 属性
    count,
    // 方法
    updateCount
  }
}
```

## ref获取页面元素

```
<template>
  <h2>ref的另一个作用:可以获取页面中的元素</h2>
  <input type="text" ref="inputRef" />
</template>
<script lang="ts">
import { defineComponent, onMounted, ref } from 'vue'
export default defineComponent({
  name: 'App',
  // 需求:当页面加载完毕后,页面中的文本框可以直接获取焦点(自动获取焦点)
  setup() {
    // 默认是空的,页面加载完毕,说明组件已经存在了,获取文本框元素
    const inputRef = ref<HTMLElement | null>(null)
    // 页面加载后的生命周期组合API
    onMounted(() => {
      inputRef.value && inputRef.value.focus() // 自动获取焦点
    })
    return {
      inputRef,
    }
  },
})
</script>
```

# reactive

定义多个数据的响应式

```
/*
const proxy = reactive(obj): 接收一个普通对象然后返回该普通对象的响应式代理器对象
```

```
响应式转换是“深层的”：会影响对象内部所有嵌套的属性
内部基于 ES6 的 Proxy 实现，通过代理对象操作源对象内部数据都是响应式的
*/
setup() {
  // const obj: any = { // 为了在使用obj.gender='男' 的时候不出现这种错误的提示信息才这么书写
  const obj = {
    name: '小明',
    age: 20,
    wife: {
      name: '小甜甜',
      age: 18,
      cars: ['奔驰', '宝马', '奥迪'],
    },
  }
  // 把数据变成响应式的数据
  // 返回的是一个Proxy的代理对象,被代理的目标对象就是obj对象
  // user现在是代理对象,obj是目标对象
  // user对象的类型是Proxy
  const user = reactive<any>(obj)
  // 方法
  // function updateUser(){}
  const updateUser = () => {
    // 直接使用目标对象的方式来更新目标对象中的成员的值,是不可能的,只能使用代理对象的方式来更新数据(响应式数据)
    // 下面的可以
    user.name += '=='
    user.age += 2
    user.wife.name += '++'
    user.wife.cars[0] = '玛莎拉蒂'
    // user---->代理对象,obj---->目标对象
    // user对象或者obj对象添加一个新的属性,哪一种方式会影响界面的更新
    obj.gender = '男' // 这种方式,界面没有更新渲染
    user.gender = '男' // 这种方式,界面可以更新渲染,而且这个数据最终也添加到了obj对象上了
    // user对象或者obj对象中移除一个已经存在的属性,哪一种方式会影响界面的更新
    delete obj.age // 界面没有更新渲染,obj中确实没有了age这个属性
    delete user.age  // 界面更新渲染了,obj中确实没有了age这个属性

    // 总结：如果操作代理对象,目标对象中的数据也会随之变化,同时如果想要在操作数据的时候,界面也要跟着重新更新渲染,那么也是操作代理对象

    // 通过当前的代理对象找到该对象中的某个属性,更改该属性中的某个数组的数据
    user.wife.cars[1] = '玛莎拉蒂'
    // 通过当前的代理对象把目标对象中的某个数组属性添加一个新的属性
    user.wife.cars[3] = '奥拓'
  }
  return {
    user,
    updateUser,
  }
},
```

# 比较Vue2与Vue3的响应式

## vue2

- 核心:
  - 对象: 通过defineProperty对对象的已有属性值的读取和修改进行劫持(监视/拦截)
  - 数组: 通过重写数组更新数组一系列更新元素的方法来实现元素修改的劫持
- 问题
  - 对象直接新添加的属性或删除已有属性, 界面不会自动更新
  - 直接通过下标替换元素或更新length, 界面不会自动更新 arr[1] = {}

## vue3

- 核心:
  - 通过Proxy(代理): 拦截对data任意属性的任意(13种)操作, 包括属性值的读写, 属性的添加, 属性的删除等...
  - 通过Reflect(反射): 动态对被代理对象的相应属性进行特定的操作
  - 文档:
    - https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Proxy
    - https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Reflect

# 计算和监视

## computed

```
setup() {
  // 定义一个响应式对象
  const user = reactive({
    // 姓氏
    firstName: '东方',
    // 名字
    lastName: '不败',
  })
  // 通过计算属性的方式,实现第一个姓名的显示

  // 计算属性的函数中如果只传入一个回调函数,表示的是get
  // 返回的是一个Ref类型的对象
  const fullName1 = computed(() => {
    return user.firstName + '_' + user.lastName
  })
  // get和set
  const fullName2 = computed({
    get() {
      return user.firstName + '_' + user.lastName
    },
    set(val: string) {
      const names = val.split('_')
      user.firstName = names[0]
      user.lastName = names[1]
    },
```

```
  })
 return {
    user,
    fullName1,
    fullName2,
  }
}
```

## watch/watchEffect

```
setup() {
  // 定义一个响应式对象
  const user = reactive({
    // 姓氏
    firstName: '东方',
    // 名字
    lastName: '不败',
  })
  // 第三个姓名:
  const fullName3 = ref('')
  // watch监视----监视指定的数据 immediate 默认会执行一次watch,deep 深度监视
  watch( user,({ firstName, lastName }) => {
      fullName3.value = firstName + '_' + lastName
    },
    { immediate: true, deep: true }
  )
  // watchEffect监视,不需要配置immediate,本身默认就会进行监视,(默认执行一次)
  watchEffect(() => {
    fullName3.value = user.firstName + '_' + user.lastName
  })

  // 监视fullName3的数据,改变firstName和lastName
  watchEffect(() => {
    const names = fullName3.value.split('_')
    user.firstName = names[0]
    user.lastName = names[1]
  })

  // watch---可以监视多个数据的
  watch([user.firstName,user.lastName,fullName3],()=>{
    // fullName3是响应式的数据,但是,user.firstName,user.lastName不是响应式的数据
  })
  // 当我们使用watch监视非响应式的数据的时候,代码需要改一下
  watch([()=>user.firstName, ()=>user.lastName,fullName3], () => {
  })
  return {
    user,
    fullName3,
  }
}
```

# 生命周期

- `beforeCreate` -> 使用 `setup()`
- `created` -> 使用 `setup()`
- `beforeMount` -> `onBeforeMount`
- `mounted` -> `onMounted`
- `beforeUpdate` -> `onBeforeUpdate`
- `updated` -> `onUpdated`
- `beforeDestroy` -> `onBeforeUnmount`
- `destroyed` -> `onUnmounted`
- `errorCaptured` -> `onErrorCaptured`

```
setup() {
  onBeforeMount(()=>{
    console.log('3.0中的onBeforeMount')
  })
  onMounted(()=>{
    console.log('3.0中的onMounted')
  })
  onBeforeUpdate(()=>{
    console.log('3.0中的onBeforeUpdate')
  })
  onUpdated(()=>{
     console.log('3.0中的onUpdated')
  })
  onBeforeUnmount(()=>{
     console.log('3.0中的onBeforeUnmount')
  })
  onUnmounted(()=>{
     console.log('3.0中的onUnmounted')
  })
}
```

# 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
- 应用: 当从合成函数返回响应式对象时，toRefs 非常有用，这样消费组件就可以在不丢失响应式的情况下对返回的对象进行分解使用
- 问题: reactive 对象取出的所有属性值都是非响应式的
- 解决: 利用 toRefs 可以将一个响应式 reactive 对象的所有原始属性转换为响应式的 ref 属性

```
function useFeatureX() {
  const state = reactive({
    name2: '自来也',
    age2: 47,
  })
  return {
    ...toRefs(state),
```

```
  }
}
export default defineComponent({
  name: 'App',
  setup() {
    const state = reactive({
      name: '自来也',
      age: 47,
    })
    // toRefs可以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
    // const state2 = toRefs(state)
    const { name, age } = toRefs(state)
    // 定时器,更新数据,(如果数据变化了,界面也会随之变化,肯定是响应式的数据)
    setInterval(() => {
      name.value += '==='
    }, 1000)
    const { name2, age2 } = useFeatureX()
    return {
      // ...state   不是响应式的数据了---->{name:'自来也',age:47}
      // ...state2  toRefs返回来的对象
      name,
      age,
      name2,
      age2,
    }
  },
})
```

# toRef

- 为源响应式对象上的某个属性创建一个 ref对象, 二者内部操作的是同一个数据值, 更新时二者是同步的
- 区别ref: 拷贝了一份新的数据值单独操作, 更新时相互不影响
- 应用: 当要将 某个prop 的 ref 传递给复合函数时，toRef 很有用

```
setup() {
  const state = reactive({
    age: 5,
    money: 100,
  })
  // 把响应式数据state对象中的某个属性age变成了ref对象了
  const age = toRef(state, 'age')
  // 把响应式对象中的某个属性使用ref进行包装,变成了一个ref对象
  const money = ref(state.money)
  const update = () => {
    // 更新数据的
    state.age += 2
  }
  return {
    state,
    age,
    money,
```

```
    update,
  }
}
=====================================================
function useGetLength(age: Ref) {
  return computed(() => {
    return age.value.toString().length
  })
}
export default defineComponent({
  name: 'Child',
  props: {
    age: {
      type: Number,
      required: true, // 必须的
    },
  },
  setup(props) {
    const length = useGetLength(toRef(props, 'age'))
    return {
      length,
    }
  },
})
```

# shallowReactive/shallowRef

- shallowReactive：只处理了对象内最外层属性的响应式(也就是浅响应式)
- shallowRef：只处理了value的响应式, 不进行对象的reactive处理
- 什么时候用浅响应式呢?
  - 一般情况下使用ref和reactive即可
  - 如果有一个对象数据, 结构比较深, 但变化时只是外层属性变化 ===> shallowReactive
  - 如果有一个对象数据, 后面会产生新的对象来替换 ===> shallowRef

```
setup() {
  // 深度劫持(深监视)----深度响应式
  const m1 = reactive({
    name: '鸣人',
    age: 20,
    car: {
      name: '奔驰',
      color: 'red',
    },
  })
  // 浅劫持(浅监视)----浅响应式
  const m2 = shallowReactive({
    name: '鸣人',
    age: 20,
```

```
    car: {
      name: '奔驰',
      color: 'red',
    },
  })
  // 深度劫持(深监视)----深度响应式----做了reactive的处理
  const m3 = ref({
    name: '鸣人',
    age: 20,
    car: {
      name: '奔驰',
      color: 'red',
    },
  })
   // 浅劫持(浅监视)----浅响应式
  const m4 = shallowRef({
    name: '鸣人',
    age: 20,
    car: {
      name: '奔驰',
      color: 'red',
    },
  })
  return {
    m1,
    m2,
    m3,
    m4,
    update,
  }
}
```

# readonly/shallowReadonly

- readonly
  - 深度只读数据
  - 获取一个对象 (响应式或纯对象) 或 ref 并返回原始代理的只读代理。
  - 只读代理是深层的：访问的任何嵌套 property 也是只读的。
- shallowReadonly
  - 浅只读数据
  - 创建一个代理，使其自身的 property 为只读，但不执行嵌套对象的深度只读转换
- 应用场景
  - 在某些特定情况下, 我们可能不希望对数据进行更新的操作, 那就可以包装生成一个只读代理对象来读取数据, 而不能修改或删除

```
setup() {
  const state = reactive({
    name: '佐助',
    age: 20,
```

```
    car: {
      name: '奔驰',
      color: 'yellow',
    },
  })
  // 只读的数据---深度的只读
  const state2 = readonly(state)
  // 只读的数据---浅只读的
  const state2 = shallowReadonly(state)
  const update = () => {
      state2.car.name += '===' //可以修改成功
  }
  return {
    state2,
    update,
  }
},
```

# toRaw/markRaw

- 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，写入时也不会触发界面更新。
- markRaw
  - 标记一个对象，使其永远不会转换为代理。返回对象本身
  - 应用场景:
    - 有些值不应被设置为响应式的，例如复杂的第三方类实例或 Vue 组件对象。
    - 当渲染具有不可变数据源的大列表时，跳过代理转换可以提高性能。

```
interface UserInfo {
  name: string;
  age: number;
  likes?: string[];
}
export default defineComponent({
  name: 'App',
  setup() {
    const state = reactive<UserInfo>({
      name: '小明',
      age: 20,
    })
    const testToRaw = () => {
      // 把代理对象变成了普通对象了,数据变化,界面不变化
      const user = toRaw(state)
      user.name += '=='
    }
    const testMarkRaw = () => {
      const likes = ['吃', '喝']
      // markRaw标记的对象数据,从此以后都不能再成为代理对象了
```

```
      state.likes = markRaw(likes)
      setInterval(() => {
        if (state.likes) {
          state.likes[0] += '='
        }
      }, 1000)
    }
    return {
      state,
      testToRaw,
      testMarkRaw,
    }
  },
})
```

# customRef

- 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式控制
- 需求: 使用 customRef 实现 debounce 的示例

```
// 自定义hook防抖的函数
// value传入的数据,将来数据的类型不确定,所以,用泛型,delay防抖的间隔时间.默认是200毫秒
function useDebouncedRef<T>(value: T, delay = 200) {
  // 准备一个存储定时器的id的变量
  let timeOutId: number
  return customRef((track, trigger) => {
    return {
      // 返回数据的
      get() {
        // 告诉Vue追踪数据
        track()
        return value
      },
      // 设置数据的
      set(newValue: T) {
        // 清理定时器
        clearTimeout(timeOutId)
        // 开启定时器
        timeOutId = setTimeout(() => {
          value = newValue
          // 告诉Vue更新界面
          trigger()
        }, delay)
      },
    }
  })
}
export default defineComponent({
  name: 'App',
  setup() {
    // const keyword = ref('abc')
```

```
    const keyword = useDebouncedRef('abc', 500)
    return {
      keyword,
    }
  },
})
```

# provide/inject

- provide 和 inject 提供依赖注入，功能类似 2.x 的 provide/inject
- 实现跨层级组件(祖孙)间通信

```
setup() {
  // 响应式的数据
  const color = ref('red')
  // 提供数据
  provide('color',color)
  return {
    color,
  }
}
================================
setup(){
  // 注入的操作
  const color = inject('color')
  return {
    color
  }
}
```

# 响应式数据的判断

- isRef: 检查一个值是否为一个 ref 对象
- isReactive: 检查一个对象是否是由 `reactive` 创建的响应式代理
- isReadonly: 检查一个对象是否是由 `readonly` 创建的只读代理
- isProxy: 检查一个对象是否是由 `reactive` 或者 `readonly` 方法创建的代理

```
setup(){
  // isRef: 检查一个值是否为一个 ref 对象
  console.log(isRef(ref({})))
  // isReactive: 检查一个对象是否是由 reactive 创建的响应式代理
  console.log(isReactive(reactive({})))
  // isReadonly: 检查一个对象是否是由 readonly 创建的只读代理
  console.log(isReadonly(readonly({})))
  // isProxy: 检查一个对象是否是由 reactive 或者 readonly 方法创建的代理
  console.log(isProxy(readonly({})))
  console.log(isProxy(reactive({})))
  return{}
}
```