做项目必会的vue3基础知识(22种)

1.响应式

1.1 两者实现原理

- vue2 利用es5的 `Object.defineProperty()` 对数据进行劫持结合发布订阅模式来实现
- vue3 利用es6的 `proxy` 对数据代理，通过 reactive() 函数给每一个对象都包一层 proxy，通过 proxy 监听属性的变化，从而实现对数据的监控

1.2 vue2响应式缺陷

缺陷

`对象新增、删除属性没有响应式`，`数组新增、删除元素没有响应式；通过下标修改某个元素没有响应式；通过.length改变数组长度没有响应式`。只有实例创建时 data 中有的数据实例创建后才是响应式的，给已创建好的 vue 实例 data 对象中添加属性时，数据虽然会更新，但视图不会更新，不具有响应式

解决

- 使用 `this.$forceUpdate()` 强制更新视图和数据（不推荐）
- 使用具有响应式的函数来操作对象：

```
[Vue | this].$set(object,key,value),实例中添加响应式属性;
[Vue | this].$delete(object,key)，实例中删除属性;
Object.assign()，将多个对象属性合并到目标对象中，具有响应式;
Object.freeze()，将对象冻结，防止任何改变。使得对象成为只读，无法添加、删除或更新;
Object.keys()，返回对象的所有属性;
Object.values()，返回对象的所有属性值;
Object.entries(),返回对象的所有键值对;
```

- 使用具有响应式的函数来操作数组：

```
pop(),尾部删除元素;
push(),尾部添加元素;
unshift(),首部添加元素;
shift(),首部删除元素;
```

```
sort(),排序;
reverse(),翻转;
splice(index[必填，位置]，howmany[必填，要删除的数量],item1...itemx[可选，向数组中添加的新元素]);
```

【补充】清空对象，置空数组操作

- 清空对象

```
this.form = {}
this.$refs.form.resetFields()
this.form.name = ""
```

- 置空数组

```
this.arrayList = []
this.arrayList.splice(0,this.arrayList.length)
// this.arrayList.length = 0  不具有响应式，无法实现
```

1.3 vue3响应式优势

- `proxy性能整体上优于Object.defineProperty`
- `vue3支持更多数据类型的劫持`（vue2只支持Object、Array；vue3支持Object、Array、Map、WeakMap、Set、WeakSet）
- `vue3支持更多时机来进行依赖收集和触发通知`（vue2只在get时进行依赖收集，vue3在get/has/iterate时进行依赖收集；vue2只在set时触发通知，vue3在set/add/delete/clear时触发通知），`所以vue2中的响应式缺陷vue3可以实现`
- `vue3做到了“精准数据”的数据劫持`（vue2会把整个data进行递归数据劫持，而vue3只有在用到某个对象时，才进行数据劫持，所以响应式更快并且占内存更小）
- `vue3的依赖收集器更容易维护`（vue3监听和操作的是原生数组；vue2是通过重写的方法实现对数组的监控）

2.生命周期

vue2	vue3	说明
beforeCreate	setup()	组件创建之前，执行初始化任务
created	setup()	组件创建完成，访问数据、获取接口数据
beforeMount	onBeforeMount	组件挂载之前
mounted	onMounted	组件挂载完成，DOM已创建，访问数据或DOM元素，访问子组件
beforeUpdate	onBeforeUpdate	未更新，获取更新前所有状态
updated	onUpdated	已更新，获取更新后所有状态
beforeDestroy	onBeforeUnmount	组件销毁之前，清空定时器，取消订阅消息
destroyed	onUnmounted	组件销毁之后
activated	onActivated	keep-alive包含，组件被激活时
deactivated	onDeactivated	keep-alive包含，发生组件切换，组件消失时

2.1 初始化

- vue2 一般在 created、mounted 中初始化
- vue3 可以直接在 setup 中，或者放在 onBeforeMount、onMounted 中初始化

```vue
<script setup>
const getList = () => {}

getList()

onMounted(() => {
  getList()
}),

onBeforeMount(() => {
  getList()
}),
</script>
```

2.2 解除绑定

- vue2 中操作：

```vue
<script>
export default {
  mounted() {
    // 开启定时器
    let timer = setInterval(() => {
      console.log('---定时器在触发---')
    }, 1000)

   //这下面的代码不可以省略
    this.$on('hook:activated', () => {
      if (timer === null) { // 避免重复开启定时器
        timer = setInterval(() => {
          console.log('setInterval')
        }, 1000)
      }
    })

    this.$on('hook:deactivated', () => {
      clearInterval(timer)
      timer = null
    })
  }
}
<script>
```

- vue3 中操作：

```vue
<script setup>
import { onBeforeUnmount, onDeactivated } from 'vue'

// 组件卸载前，对应 Vue2 的 beforeDestroy
onBeforeUnmount(() => {
    clearTimeout(timer)
    window.removeAddEventListener('...')
})

// 退出缓存组件，对应 Vue2 的 deactivated
onDeactivated(() => {
    clearTimeout(timer)
    window.removeAddEventListener('...')
})
</script>
```

3.this指向

- vue2中可以调用this来指向当前实例，this上挂载了路由、状态管理、公共的组件、方法等可以访问、使用
- 通过上面的生命周期可以看出来，vue3中 `setup()` 在解析其他组件选项（data、methods、computed 等都没解析）之前调用，在beforeCreate（）之前执行，所以this指向undefined，vue3中不能通过this进行访问
- vue3想要执行类似vue2调用this的用法可以进行如下操作：

```vue
<script setup>
import { getCurrentInstance } from "vue";

// proxy 为当前组件实例;global 为全局组件实例
const { proxy, appContext } = getCurrentInstance();
const global = appContext.config.globalProperties;
</script>
```

4.变量

4.1 ref

- ref 定义 `基本类型` 生成 `RefImpl` 实例；定义 `复合类型` 生成 `Proxy` 实例
- template 渲染直接使用，js中修改通过 `.value` 调用

```
const count = ref(0)
const user = ref({
    name:'falcon',
    age:20
})

const addCount = () => count.value++
const addAge = () => user.value.age++
```

4.2 reactive

- reactive 只能定义对象类型的数据，生成 `Proxy` 实例
- template、js中可以直接调用
- `shallowReactive` 生成非递归响应数据，只监听第一层数据的变化

```
const stu = reactive({
    name:'falcon',
    major:'Chinese',
    score:80
})

const addScore = () => stu.score++
```

4.3 转化响应式

- `toRef()`，单个转化为响应式
- `toRefs()`，多个转化为响应式
- `unref()`，是 `val = isRef(val) ? val.value : val` 的语法糖；如果参数是一个ref就返回其 value，否则返回参数本身

【注】针对一个响应式对象（reactive封装）的prop（属性）创建一个ref，且保持响应式

```
const stu = reactive({
    name:'falcon',
    age:20,
    major:'Chinese',
    score:80
})
const age = toRef(stu,'age')
const {name,major,score} = toRefs(stu)
```

4.4 只读

- `readonly`，创建只读对象（递归只读）
- `isReadonly`，判断是否是readonly对象
- `shallowReadonly`，只对最外层响应式只读，深层次不转换

```
let status = readonly(true);
const changeStatus = () => (status = !status);

let info = reactive({
  username: "falcon",
  password: "123456",
  role: {
    roleId: 123,
    roleName: "系统管理员",
  },
});
info = shallowReadonly(info);
const changeRole = () => {
  info.role.roleId++;
};
```

5.Fragment

- vue2 中 `只能有一个根节点`，因为vdom是一颗单根树，patch方法在遍历的时候从根节点开始，所以要求template只有一个根元素
- vue3 中 `可以有多个根节点`，因为如果template不只有一个根元素时，就会添加一个fragment组件将多个根组件包起来

```
<template>
  <div>demo1 text</div>
  <h2>h2 text</h2>
  <p>p text</p>
</template>
```

6.Teleport

- teleport 瞬移组件，能将我们的元素移动到DOM中vue app之外的其他位置（有时用于页面需要弹框且弹框不影响布局的情况，相对于body进行定位）

```vue
<template>
  <div class="app-container">
    <el-button type="primary" @click="showToast">打开弹框</el-button>
  </div>
  <teleport to="body">
    <div v-if="visible" class="modal_class">
      A man who has not climbed the granted wall is not a true man
      <el-button
        style="width: 50%; margin-top: 20px"
        type="primary"
        @click="closeToast"
        >关闭弹框</el-button
      >
    </div>
  </teleport>
</template>

<script setup>
import { ref } from "vue";

const visible = ref(false);
const showToast = () => {
  visible.value = true;
};
const closeToast = () => {
  visible.value = false;
};
</script>

<style scoped>
.modal_class {
  position: absolute;
  width: 300px;
  height: 200px;
  top: 50%;
  left: 50%;
  transform: translate(-50%, -50%);
  border: 1px solid #ccc;
  display: flex;
  flex-direction: column;
  justify-content: center;
  align-content: center;
  padding: 30px;
```

```
}
</style>

```

7.Suspense

- suspense 允许程序在等待一些异步组件时加载一些后备内容
- 在异步加载请求网络数据时，使用suspense组件，可以很好的实现loading效果
- `#default` 初始化模板组件； `#fallback` 异步请求中处理的ui

```
<template>
  <div class="app-container">
    <Suspense>
      <template #default>
        <SyncApi />
      </template>
      <template #fallback>
        <h3 style="color: blue">数据加载中...</h3>
      </template>
    </Suspense>
  </div>
</template>

<script setup>
import SyncApi from "./SyncApi.vue";
</script>
```

```
// SyncApi 组件内容
<template>
  <div v-for="(people, index) in peoples.results" :key="index">
    {{ people.name }} {{ people.birth_year }}
  </div>
</template>

<script setup>
const peoples = ref({
  results: [],
});
const headers = { "Content-Type": "application/json" };
const fetchPeoples = await fetch("https://swapi.dev/api/people", {
  headers,
```

```
});
peoples.value = await fetchPeoples.json();
</script>
```

8.组件

- vue2 中组件引入后需在components中注册后才能使用
- vue3 中组件引入后直接使用无需注册

```
<template>
  <Table />
</template>

<script setup>
import Table from "@/components/Table";
</script>
```

9.获取DOM

```
<template>
  <el-form ref="formRef"></el-form>
</template>

<script setup>
// 1. 变量名和 DOM 上的 ref 属性必须同名，自动形成绑定
const formRef = ref(null)
console.log(formRef.value)

// 2. 通过当前组件实例来获取DOM元素
const { proxy } = getCurrentInstance()
proxy.$refs.formRef.validate((valid) => { ... })
</script>
```

10.watch、watchEffect

10.1 watch

```
// vue2 中用法
```

```
watch:{
    // 第一种
    flag(newValue,oldValue){},

    // 第二种
    user:{
        handler(newValue,oldValue){},
        immediate:true,
        deep:true
    }
}
```

```
// vue3 中用法
<script setup>
const count = ref(0)
const status = ref(false)

// 监听一个
watch(count,(newValue,oldValue) => {})
// 监听多个
watch([count,status],([newCount,oldCount],[newStatus,oldStatus]) => {})

const user = reactive({
    name:'falcon',
    age:20,
    sex:'female',
    hobbies:[]
})

// 监听一个
watch(() => user.age,(newValue,oldValue) => {})
// 监听多个
watch([() => user.name,() => user.sex],(newValue,oldValue) => {})

// 添加配置参数
watch(() => user.hobbies,(newValue,oldValue)=> {},{
    immediate:true,
    deep:true,
    // 回调函数的执行时机，默认在组件更新之前执行，更新之后执行参数为‘post’
    flush:'pre'
})
</script>
```

10.2 watchEffect

```
// 正常情况组件销毁自动停止监听
watchEffect(() => {})

// 异步方式手动停止监听
const stopWatch = watch(() => user.hobbies,(newValue,oldValue)=>{},{deep:true})
setTimeout(() => {
    stopWatch()
},3000)

const stopWatchEffect = watchEffect(() => {})
setTimeout(() => {
    stopWatchEffect()
},3000)
```

10.3 两者区别

- watch `对传入的一个或多个值进行监听`，触发时会 `返回新值和旧值`，且 `默认第一次不会执行`
- watchEffect 是传入一个立即执行函数，`默认第一次会执行，且不需要传入监听的内容，会自动收集函数内的数据源作为依赖`，当依赖发生变化时会重新执行函数（类似computed），并且 `不会返回旧值`
- 正常情况下，组件销毁/卸载后这两种方式都会停止监听，但是 `异步方式` 例如setTimeout里创建的监听需要手动停止

11.computed

- 默认只改变数据的值，如果想改变后的值 `具有响应性`，利用其 `set()` 方法
- vue3.x中移除过滤器filter，建议使用computed
- `回调函数必须return`，结果是计算结果
- 计算属性依赖的 `数据项发生变化时，重新计算，具有缓存性`
- `不能执行异步操作`

```
const names = reactive({
    firstName:'',
    lastName:'',
    fullName:''
})
```

```
6
7  // 通过此种方式定义的fullName,想要修改的时候后台警告：Write operation failed:
   computed value is readonly; 想要修改fullName，通过set()方法
8  const fullName = computed(() => {
9    return names.firstName + " " + names.lastName;
10 });
11
12 const fullName = computed({
13     get(){
14         return names.firstName + " " + names.lastName
15     },
16     set(value){
17
18     }
19 })
```

12.可组合函数

- vue3 中组合式api `使用hooks代替 vue2 中的mixin`。hooks 约定用驼峰命名法，并以“use”作为开头；将可复用的功能抽离为外部js文件；引用时将定义的属性和方法响应式地解构暴露出来；避免了vue2的碎片化，实现低内聚高耦合
- 传统mixin的缺陷：
  - mixin可能会引起命名冲突和重复代码，后期很难维护
  - mixin可能会导致组件之间的依赖关系不清楚，不好追溯源
- mixin生命周期函数：`先执行mixin中生命周期函数；后执行组件内部代码`，`mixin中的data数据和组件中的data数据冲突时，组件中的data数据会覆盖mixin中数据`

```
1  // useCount.js
2  const useCount = (initValue = 1) => {
3      const count = ref(initValue)
4
5      const increase = (delta) => {
6          if(typeof delta !== 'undefined'){
7              count.value += delta
8          }else{
9              count.value++
10         }
11     }
12
13     const multiple = computed(() => count.value * 2)
14
15     const decrease = (delta) => {
```

```
        if(typeof delta !== 'undefined'){
            count.value -= delta
        }else{
            count.value--
        }
    }

    const reset = () => count.value = initValue

    return {
        count,
        multiple,
        increase,
        decrease,
        reset
    }
}

export default useCount
```

```
<template>
    <p>{{count}}</p>
    <p>{{multiple}}</p>
    <el-button @click="addCount">count++</el-button>
    <el-button @click="subCount">count--</el-button>
    <el-button @click="resetCount">reset</el-button>
</template>

<script setup>
import useCount from "@/hooks/useCount"
const {count,multiple,increase,decrease,reset} = useCount(10)
const addCount = () => increase()
const subCount = () => decrease()
const resetCount = () => reset()
</script>
```

13.懒加载组件

```
// Demo.vue
<template>
    <div>异步加载组件的内容</div>
</template>
```

```vue
// ErrorComponent.vue
<template>
    <div>Warning: 组件加载异常</div>
</template>
```

```vue
// LoadingComponent.vue
<template>
    <div>组件正在加载...<div>
</template>
```

```vue
<template>
    <AsyncDemo />
</template>

<script setup>
import LoadingComponent from './LoadingComponent.vue'
import ErrorComponent from './ErrorComponent.vue'

const time = (t,callback = () => {}) => {
    return new Promise((resolve) => {
        setTimeout(() => {
            callback()
            resolve()
        },t)
    })
}

const AsyncDemo = defineAsyncComponent({
    // 要加载的组件
    loader:() => {
        return new Promise((resolve) => {
            async function(){
                await time(3000)
                const res = await import("./Demo.vue")
                resolve(res)
            }
        })
    },
    // 加载异步组件时使用的组件
    loadingComponent:LoadingComponent,
    // 加载失败时使用的组件
```

```
    errorComponent:ErrorComponent,
    // 加载延迟（在显示loadingComponent之前的延迟），默认200
    delay:0,
    // 超时显示组件错误，默认永不超时
    timeout:5000
})
</script>
```

14.插槽

- 具名插槽使用方式不同：vue2 中使用 `slot='插槽名称'` ，vue3 中使用 `v-slot:插槽名称`
- 作用域插槽使用方式不同：vue2 中在父组件中使用 `slot-scope="data"` 从子组件获取数据，vue3 中在父组件中使用 `#data` 或者 `#default="{data}"` 获取

```
<template>
    <div>
        <!-- 默认 -->
        <slot />
        <!-- 具名 -->
        <slot name="slotName" />
        <!-- 作用域 -->
        <slot :data="user" name="propsSlot" />
    </div>
</template>

<script>
const user = reactive({
    name:'falcon',
    age:20
})
</script>
```

```
<template>
    <Son>
        <template #default><div>默认插槽内容</div></template>
        <template #slotName><div>具名插槽内容</div></template>
        <template #propsSlot="scope">
            <div>
                作用域插槽内容：name,{{scope.data.name}};age,{{scope.data.age}}
            </div>
        </template>
    </Son>
```

```
</template>

<script setup>
import Son from './Son.vue'
<script>
```

15.自定义指令

- 全局自定义指令在main.js中定义
- 局部自定义指令在当前组件中定义

```
// main.js
app.directive("focus",{
    mounted(el,bingings,vnode,preVnode){
        el.focus()
    }
})
```

```
<template>
    <div>
        <input type="text" v-focus />
    </div>
</template>

<script setup>
const vFocus = {
    mounted:(el) => el.focus()
}
</script>
```

16.v-model

- vue2 中 `.sync` 和 `v-model` 都是语法糖，都可以实现父子组件中数据的双向通信
- vue2两种格式差别： `v-model="num"` , `:num.sync="num"` ;v-model:@input+value,:num.sync:@update:num
- vue2中 `v-model只能用一次，.sync可以有多个`
- vue3中取消了 .sync，合并到v-model， `vue3中v-model可以有多个`

```vue
<template>
    <p>name:{{name}}</p>
    <p>age:{{age}}</p>
    <Son v-model:name="name" v-model:age="age" />
</template>

<script setup>
import Son from './Son.vue'

const user = reactive({
    name:'falcon',
    age:20
})

const {name,age} = toRefs(user)
</script>
```

```vue
<template>
    <input type="text" :value="name" @input="onNameInput" />
    <input type="number" :value="age" @change="onAgeInput" />
</template>

<script setup>
defineProps({
    name:{
        type:String,
        default:() => ""
    },
    age:{
        type:String,
        default:() => ""
    }
})

const emit = defineEmits(["update:name"],["update:age"])

const onNameInput = (e) => emit("update:name",e.target.value)
const onAgeInput = (e) => emit("update:age",e.target.value)
</script>
```

17.v-if/v-for

- 不建议 v-for 与 v-if 一起使用

- `vue2 中优先级v-for 高于 v-if`。如果执行过滤列表项操作，配合computed；如果条件判断来显隐循环列表，将v-if提前，包裹v-for
- `vue3 中优先级v-if 高于 v-for`

```
<template>
    <div v-if="flag">
        <div v-for="item in dataList" :key="item.id">{{item.id}} - {{item.label}}</div>
    </div>
</template>

<script setup>
const flag = ref(true)
const dataList = reactive([
    {
        id:1,
        label:'list-01'
    },
    {
        id:2,
        label:'list-02'
    }
])
</script>
```

18.v-bind

- vue2 中 `单独声明优先`，并且重复定义会出发出警告
- vue3 中绑定值是否生效遵循 `就近原则`

```
<template>
    <div>
        <input type="text" v-bind:disabled="false" :disabled="disabled" />
        <input type="text" :disabled="disabled" v-bind:disabled="false" />
    </div>
</template>

<script setup>
const disabled = ref(true)
</script>
```

19.组件通信

19.1 props/$emit

父组件传值，子组件通过props接受；子组件想改变父组件中数值，通过$emit调用父组件中方法

- 父组件

```
<template>
  <Child :count="count" :name="name" :age="age" @add="add" @sub="sub" />
</template>

<script setup>
import Child from "./Child.vue";

const count = ref(0);
const user = reactive({
  name: "falcon",
  age: 20,
});

const add = () => count.value++;
const sub = () => count.value--;

const { name, age } = toRefs(user);
</script>
```

- 子组件

```
<template>
  <p>接受到的参数为：name,{{ name }},age,{{ age }},count,{{ count }}</p>
  <el-button type="primary" size="small" @click="add">count++</el-button>
  <el-button type="primary" size="small" @click="sub">count--</el-button>
</template>

<script setup>
defineProps({
  name: {
    type: String,
    default: () => "",
  },
  age: {
    type: Number,
    default: () => 0,
```

```
  },
  count: {
    type: Number,
    default: () => 0,
  },
});

const emits = defineEmits(["add", "sub"]);

const add = () => emits("add");
const sub = () => emits("sub");
</script>
```

19.2 attrs

传递属性或方法给子组件下级组件，传递子组件中没有被props定义的属性，传递子组件中没有被emits定义的方法

- 父组件

```
<template>
  <Child :count="count" :name="name" :age="age" @add="add" @sub="sub" />
</template>

<script setup>
import Child from "./Child.vue";

const count = ref(0);
const user = reactive({
  name: "falcon",
  age: 20,
});

const add = () => count.value++;
const sub = () => count.value--;

const { name, age } = toRefs(user);
</script>
```

- 子组件

```
<template>
  <p>子组件接收：{{ count }}</p>
```

```
  <el-button type="primary" size="small" @click="add">count++</el-button>
  <GrandChild v-bind="$attrs" />
</template>

<script setup>
import GrandChild from "./GrandChild.vue";

defineProps({
  count: {
    type: Number,
    default: () => 0,
  },
});

const emits = defineEmits(["add"]);

const add = () => emits("add");
</script>
```

- 孙组件

```
<template>
  <p>孙组件接受：name，{{ name }},age,{{ age }}</p>
  <el-button type="primary" size="small" @click="sub">count--</el-button>
</template>

<script setup>
defineProps({
  name: {
    type: String,
    default: () => "",
  },
  age: {
    type: Number,
    default: () => 0,
  },
});

const emits = defineEmits(["sub"]);

const sub = () => emits("sub");
</script>
```

19.3 v-model

- 父组件

```
<template>
  <Child v-model:name="name" v-model:count="count" v-model:salary="salary" />
</template>

<script setup>
import Child from "./Child.vue";

const name = ref("falcon");
const count = ref(0);
const salary = ref(3000);
</script>
```

- 子组件

```
<template>
  <p>
    子组件接受到的v-model参数: name,{{ name }},count,{{ count }},salary,{{
      salary
    }}
  </p>
  <el-button type="primary" size="small" @click="changeCount"
    >count++</el-button
  >
  <el-button type="primary" size="small" @click="changeSalary"
    >salary1000+</el-button
  >
</template>

<script setup>
const props = defineProps({
  name: {
    type: String,
    default: () => "",
  },
  count: {
    type: Number,
    default: () => "",
  },
  salary: {
    type: Number,
    default: () => "",
  },
```

```
});

const emits = defineEmits(["update:count", "update:salary"]);
const changeCount = () => emits("update:count", props.count + 1);
const changeSalary = () => emits("update:salary", props.salary + 1000);
</script>
```

19.4 ref/expose

通过ref获取指定的DOM元素或组件，结合defineExpose暴露出来的属性和方法实现通信

- 父组件

```
<template>
  <div>title:{{ title }}</div>
  <Child ref="child" />
  <el-button type="primary" size="small" @click="add">count++</el-button>
  <el-button type="primary" size="small" @click="sub">count--</el-button>
  <el-button type="primary" size="small" @click="receive"
    >receive msg</el-button
  >
</template>

<script setup>
import Child from "./Child.vue";

const child = ref(null);
const title = ref("暂无数据");
const add = () => child.value.add();
const sub = () => child.value.sub();
const receive = () => (title.value = child.value.msg);
</script>
```

- 子组件

```
<template>
  <p>子组件: count,{{ count }}</p>
</template>

<script setup>
const count = ref(0);
const msg = "expose message";
const add = () => count.value++;
const sub = () => count.value--;
```

```
10
11 defineExpose({
12   msg,
13   add,
14   sub,
15 });
16 </script>
```

19.5 provide/inject

祖先向下级传递参数，无论层级多深，都可以传递

- 父组件

```
1 <template>
2   <Child />
3 </template>
4
5 <script setup>
6 import Child from "./Child.vue";
7
8 const user = reactive({
9   name: "falcon",
10   age: 20,
11 });
12 provide("user", user);
13 </script>
```

- 子组件

```
1 <template>
2   <p>子组件接受：name,{{ user.name }}</p>
3   <GrandChild />
4 </template>
5
6 <script setup>
7 import GrandChild from "./GrandChild.vue";
8
9 const user = inject("user");
10 </script>
```

- 孙组件

```vue
<template>
  <p>孙组件接受: age,{{ user.age }}</p>
</template>

<script setup>
const user = inject("user");
</script>
```

19.6 mixin

不建议使用，建议使用可组合函数完成组件间通信和复用

- mixin.js

```vue
const count = ref(20);
const name = ref("falcon");

const addCount = () => count.value++;
const subCount = () => count.value--;

export default { count, name, addCount, subCount };
```

- 组件使用

```vue
<template>
  <p>name:{{ name }},count:{{ count }}</p>
  <el-button @click="addCount" type="primary" size="small">count++</el-button>
  <el-button @click="subCount" type="primary" size="small">count--</el-button>
</template>

<script setup>
import mixins from "./mixin";
const { count, name, addCount, subCount } = mixins;
</script>
```

19.7 mitt

vue3 中废除api: `$on` 、 `$once` 、 `$off` ;不再支持Event Bus，选用替代方案mitt.js,原理还是Event Bus

- bus.js

```
import mitt from 'mitt';
export default mitt()
```

- 父组件

```
<template>
  <Brother1 />
  <Brother2 />
</template>

<script setup>
import Brother1 from "./Brother1.vue";
import Brother2 from "./Brother2.vue";
</script>
```

- 兄弟组件1

```
<template>
    <p>brother1 发送事件</p>
    <el-button type="primary" size="small" @click="handleClick">发送事件</el-button>
</template>

<script setup>
import mybus from './bus.js';

const handleClick = () => {
    mybus.emit("title",{title:"hello world"});
    mybus.emit("user",{user:{name:"falcon",age:20}})
}
</script>
```

- 兄弟组件2

```
<template>
    <p>brother2 接受事件</p>
    <p>title:{{title}}</p>
    <p>user:name,{{name}};age,{{age}}</p>
</template>

<script setup>
```

```
import mybus from './bus.js'

const title = ref("")
const user = reactive({
    name:"",
    age:null
})

mybus.on("title",(data) => {
    title.value = data.title
})
mybus.on("user",(data) => {
    user.name = data.user.name
    user.age = data.user.age
})
</script>
```

20.状态管理 pinia

pinia 是 vue 的存储库，允许 `跨组件/跨页面共享状态` 。具有以下优点：

- 轻量，约1kb
- `去除Mutation，Actions支持同步和异步`
- 无需手动注册store， `store仅在需要时才自动注册`
- 没有模块嵌套，store之间可以自由使用
- `支持模块热更新`

20.1 创建

```
import { createPinia } from 'pinia'

const store = createPinia()
export default store
```

20.2 定义

```
// 引入store定义函数
import { defineStore } from 'pinia'

// 定义store实例并导出
```

```
// 第一个参数，字符串类型，唯一不可重复，作为库id来区分不同库
// 第二个参数，以对象形式配置存储库的state、getters、actions

export const useStore = defineStore('useCount',{
    /**
        state,存储全局状态
        必须是箭头函数：为了在服务器端渲染的时候避免交叉请求导致数据状态污染
    */
    state:() => {
        return {
            count:0
        }
    },
    /**
        getters,封装计算属性
        具有缓存功能，类似于computed;需要传入state才能拿到数据;不能传递任何参数，但是可以返回一个函数接受任何参数
    */
    getters:{
        doubleCount:(state) => state.count * 2,
        powCount(){
            return this.doubleCount ** 2
        }
    },
    /**
        actions,编辑业务逻辑
        类似于methods，支持同步和异步；获取state里的数据不需要传入直接使用this
    */
    actions: {
        addCount(){
            this.count++
        },
        subCount(){
            this.count--
        }
    },
    /**
        配置数据持久化需要进行的操作
    */
    persist:{}
})
```

20.3 页面使用

```
<template>
```

```
    <p>{{useStoreDemo.count}}<p>
    <p>{{useStoreDemo.doubleCount}}</p>
    <p>{{useStoreDemo.powCount}}</p>
    <el-button @click="toAdd">count++</el-button>
    <el-button @click="toSub">count--</el-button>
</template>

<script setup>
import {useStore} from '../store'
const useStoreDemo = useStore()

// 也可以解构出来想要使用的count，但直接解构不具有响应式，想要具有响应式，可以执行如下操作:
const {count} = storeToRefs(useStore())

const toAdd = () => useStoreDemo.addCount()
const toSub = () => useStoreDemo.subCount()
<script>
```

20.4 数据持久化

pinia的数据是存储在内存中的，页面刷新后数据会丢失；可以支持扩展插件，实现数据持久化

- `npm i pinia-plugin-persist`,默认使用sessionStorage
- 配置使用代码如下：

```
persist:{
    enabled:true,
    strategies:[
        {
            storage:localStorage,
            paths:["num","user"]
        }
    ]
}
```

21.路由

- query传参配置path，params传参配置name，且`params中配置path无效`
- query传参显示在地址栏，params传参不会
- `query传参刷新页面数据不会消失，params传参刷新页面数据消失`

- params可以使用动态参数（"/path/:params"），`动态参数会显示在地址栏中，且刷新页面数据不会消失`
- name为路由中定义的name属性，`严格区分大小写`
- 路由跳转：前进`router.go(1)`、后退`router.go(-1)`、刷新`router.go(0)`
- 使用案例：

```
<template>
    <el-button @click="TransByQuery">通过query传参</el-button>
    <el-button @click="TransByParams">通过params传参</el-button>
    <el-button @click="TransByDynamic">动态传递参数</el-button>
</template>

<script setup>
const queryParams = reactive({
    name:'falcon',
    age:20
})

const id = ref('2023')

const router = useRouter()

const TransByQuery = () => {
    router.push({
        path:'/basic/querydemo',
        query:queryParams
    })
}

const TransByParams = () => {
    router.push({
        name:'ParamsDemo',
        params:queryParams
    })
}

const TransByDynamic = () => {
    router.push({
        name:'DynamicDemo',
        params:{id:id.value}
    })
}
<script>
```

- query 接受参数

```
const route = useRoute()
console.log(route.query.name,route.query.age)
```

- params 接受参数

```
const route = useRoute()
console.log(route.params.name,route.params.age)
```

- 动态传递 接受参数

```
const route = useRoute()
console.log(route.params.id)
```

- 相应的路由

```
 {
  name: "QueryDemo",
  path: "querydemo",
  redirect: null,
  component: "basic/userouter/querydemo",
  hidden: true,
  meta: {
    title: "query样例",
    icon: null,
  },
},
{
  name: "ParamsDemo",
  path: "paramsdemo",
  redirect: null,
  component: "basic/userouter/paramsdemo",
  hidden: true,
  meta: {
    title: "params样例",
    icon: null,
  },
},
{
```

```
    name: "DynamicDemo",
    path: "dynamicdemo/:id",
    redirect: null,
    component: "basic/userouter/dynamicdemo",
    hidden: true,
    meta: {
      title: "dynamic样例",
      icon: null,
    },
},
```

22.css补充

22.1 样式穿透

- css `className`，less `/deep/ className`,scss `::v-deep className`
- vue3中css使用：`:deep(className)`

22.2 绑定变量

```
<template>
    <div class="name">falcon</div>
</template>

<script setup>
const str = ref('#f00')
</script>

<style lang="scss" scoped>
.name{
    background-color:v-bind(str)
}
</style>
```