

# 前端VUE框架

## 1 前端中的template标签

### 什么是 `<template>` 标签?

`<template>` 标签是一个原生的 HTML 元素，用来包含一些你希望在页面上动态插入或重复使用的 HTML 代码。这些代码不会在页面加载时立即显示，而是作为一个“模板”存储在文档中，等待被 JavaScript 提取并插入到页面中。

### `<template>` 标签的主要特点

#### 1. 不会渲染到页面:

- `<template>` 标签中的内容不会在页面加载时被显示。浏览器会跳过渲染它里面的内容，用户在页面上看不到这些内容。
- 在 HTML 中，通常你写的内容会直接显示在网页上。比如，你写 `<p>Hello, world!</p>`，网页上就会显示“Hello, World!”。

但是，`<template>` 标签是一个特殊的标签，它里面的内容不会直接显示在网页上。也就是说，你写在 `<template>` 标签里的东西，用户在页面加载时是看不到的，除非你用 JavaScript 手动把它插入到页面中。

#### 2. 可以被 JavaScript 动态使用:

- 你可以通过 JavaScript 获取 `<template>` 中的内容，然后将它动态插入到页面的其他部分。这样，你可以在需要时重复使用这些 HTML 结构。

### 为什么需要 `<template>` 标签?

在一些情况下，我们希望在页面上动态地创建和插入一些 HTML 内容，比如添加新的列表项、生成表格行、创建弹出框等等。手动写这些内容不仅麻烦，还不易维护。使用 `<template>` 标签可以让你预先定义好这些内容，然后在需要时通过 JavaScript 进行插入，提升开发效率和代码可读性。

准确地说，使用 `<template>` 标签的主要目的是为了动态控制页面内容的显示。通过将不需要立即显示的内容放在 `<template>` 标签中，你可以在需要时通过 JavaScript 动态插入这些内容，而不是让所有内容都在页面加载时显示。

假设你有一个按钮，每次点击按钮时，你希望在列表中添加一个新的列表项。这时，你可以使用 `<template>` 标签来定义这个列表项的模板。

## 2 route路由 Vue3框架用于管理页面跳转

### 例子：视频播放网站的页面导航

假设我们正在开发一个简单的视频播放网站，它有三个主要页面：

1. 首页 (HomePage)：显示一个视频列表。
2. 视频播放页 (VideoPage)：当用户点击某个视频时，跳转到这个页面播放视频。
3. 关于页 (AboutPage)：显示网站的相关信息。

## 目录结构

假设我们的项目目录结构如下：

```
src/
  views/
    HomePage.vue
    VideoPage.vue
    AboutPage.vue
  App.vue
  main.js
  router.js
```

### 1. 配置路由

首先，我们在 `router.js` 文件中配置路由：

```
import { createRouter, createWebHistory } from 'vue-router';
import HomePage from './views/HomePage.vue';
import videoPage from './views/videoPage.vue';
import AboutPage from './views/AboutPage.vue';

const routes = [
  { path: '/', name: 'home', component: HomePage },
  { path: '/video/:id', name: 'video', component: VideoPage },
  { path: '/about', name: 'about', component: AboutPage }
];

const router = createRouter({
  history: createWebHistory(),
  routes
});

export default router;
```

### 2. 使用路由器

在 `main.js` 文件中，我们导入并使用这个路由器：

```
import { createApp } from 'vue';
import App from './App.vue';
import router from './router';

const app = createApp(App);
app.use(router);
app.mount('#app');
```

### 3. 创建组件

#### HomePage.vue

在首页，我们展示一组视频的列表，并为每个视频提供一个链接，点击后跳转到视频播放页：

```
<template>
  <div>
    <h1>视频列表</h1>
    <ul>
      <li v-for="video in videos" :key="video.id">
        <!-- 使用 router-link 实现跳转 -->
```

```
<router-link :to="`/video/${video.id}`">{{ video.title }}</router-link>
</li>
</ul>
<router-link to="/about">关于我们</router-link>
</div>
</template>

<script>
export default {
  data() {
    return {
      videos: [
        { id: 1, title: 'Vue.js 基础教程' },
        { id: 2, title: 'Vue Router 教程' },
        { id: 3, title: 'Vuex 状态管理' }
      ]
    };
  }
};
</script>
```

### VideoPage.vue

当用户点击视频链接时，将跳转到这个视频播放页面：

```
<template>
<div>
  <h1>正在播放: {{ videoTitle }}</h1>
  <p>视频 ID: {{ videoId }}</p>
  <!-- 实际应用中，这里会放置视频播放器 -->
</div>
</template>

<script>
export default {
  computed: {
    videoId() {
      return this.$route.params.id;
    },
    videoTitle() {
      // 根据 videoId 动态获取视频标题
      const videos = {
        1: 'Vue.js 基础教程',
        2: 'Vue Router 教程',
        3: 'Vuex 状态管理'
      };
      return videos[this.videoId] || '未知视频';
    }
  }
};
</script>
```

### AboutPage.vue

这是一个简单的关于页面：

```
<template>
<div>
  <h1>关于我们</h1>
  <p>这是一个演示 Vue Router 的简单视频播放网站。</p>
</div>
</template>
```

#### 4. 如何工作

- **首页 (HomePage.vue) :**
  - 用户看到一个视频列表，每个视频旁边都有一个链接。当用户点击链接时，浏览器的 URL 会变成 `/video/1`、`/video/2` 等等（根据视频的 ID）。
  - 这些链接是通过 `<router-link>` 组件生成的，它负责管理点击事件并触发 Vue Router 进行页面跳转，而不会刷新整个页面。
- **视频播放页面 (VideoPage.vue) :**
  - 当用户点击某个视频链接时，Vue Router 会拦截这个点击事件，读取 URL 中的 `id` 参数（如 `/video/1` 中的 `1`），然后加载 `VideoPage.vue` 组件并将该 `id` 传递给组件。
  - 在 `VideoPage.vue` 组件中，我们使用 `this.$route.params.id` 来获取当前 URL 中的 `id`，并根据这个 `id` 显示对应的视频内容。
- **关于页面 (AboutPage.vue) :**
  - 用户点击“关于我们”链接时，Vue Router 会将用户导航到 `/about` 路径，加载并显示 `AboutPage.vue` 组件。

#### 总结

- `<router-link>`：用于生成导航链接，点击后触发 Vue Router 进行页面切换，而不刷新整个页面。
- `$route.params`：用于获取当前路由的动态参数（如视频 ID）。
- **页面导航**：当用户点击不同的链接时，Vue Router 会根据配置的路由规则加载相应的组件，更新页面内容。  
通过这个例子，你可以看到 Vue Router 如何管理页面之间的导航，如何根据不同的 URL 显示不同的组件，以及如何实现无刷新页面跳转。

## 3 Ref和Reactive VUE框架创建对比

宏观角度看：

1. `ref`用来定义：基本类型数据、对象类型数据；

2. `reactive`用来定义：对象类型数据。

- 区别：

1. `ref`创建的变量必须使用 `.value`（可以使用 `volar` 插件自动添加 `.value`）。

2. `reactive`重新分配一个新对象，会失去响应式（可以使用 `Object.assign` 去整体替换）。

- 使用原则：

1. 若需要一个基本类型的响应式数据，必须使用 `ref`。

2. 若需要一个响应式对象，层级不深，`ref`、`reactive`都可以。

3. 若需要一个响应式对象，且层级较深，推荐使用 `reactive`。

## 4 直接监视和函数式监视 (Vue-watch)

### 1. 直接监视 (值传递)

当你直接传递一个响应式对象或其属性时，实际上你是将该对象或属性的当前值传递给 `watch` 函数。Vue 只会在这个值（或对象的引用）发生变化时触发回调。

```
// 直接监视对象
watch(person.value.name, (newValue, oldValue) => {
  console.log('person.name 变化', newValue, oldValue);
});
```

#### 编程语法角度

- **值传递**: 这里传递给 `watch` 的是 `person.value.name` 的当前值。`watch` 会在初次调用时读取这个值，并仅在 Vue 监测到这个具体的值发生变化（对于对象类型是其引用变化）时触发回调。
- **缺点**: 如果这个值是一个基本类型（如字符串、数字），而且你想监视它的变化，直接传递可能导致 Vue 不能正确捕捉到变化，因为 Vue 只会监视初次传递的值。

### 2. 函数式监视 (引用传递)

当你使用函数形式传递时，传递给 `watch` 的是一个函数，Vue 会在每次渲染时调用这个函数，并监视函数返回值的变化。

```
// 函数式监视对象属性
watch(() => person.value.name, (newValue, oldValue) => {
  console.log('person.name 变化', newValue, oldValue);
});
```

#### 编程语法角度

- **引用传递**: 这里传递给 `watch` 的是一个函数的引用，Vue 每次会调用这个函数来获取返回值，并监视这个返回值的变化。函数的返回值可以是任何类型：基本类型、对象、计算属性等。
- **优势**: 使用函数形式可以确保每次调用 `watch` 时，Vue 都会动态计算函数返回的值，从而能够监视对象内部属性的变化，甚至是一些复杂的计算结果。

### 区别

- 直接监视
  - 传递的是值（或引用），Vue 仅在检测到这个值本身发生变化时才触发回调。
  - 对于基本类型的值，可能不能准确捕捉到所有变化。
- 函数式监视
  - 传递的是一个函数引用，Vue 会监视函数返回的值。通过函数，可以精细控制监视的内容（如对象的某个属性或计算结果）。
  - 更适合监视对象内部的具体属性或动态计算的结果，因为每次函数返回值变化时都会触发回调。

### 总结

从语法上看，直接监视和函数式监视的区别在于参数的类型：直接监视传递的是具体的值或对象引用，而函数式监视传递的是一个函数引用。函数式监视通过动态调用函数来获取需要监视的值，从而更灵活地处理复杂的监视需求。

## 5 Ref标签的自动绑定

这里的title1 title2是通过变量名来自动和h1 h2实现绑定的  
但是这样很不直观 如果有必要 可以通过ref手动绑定 具体代码问gpt就行

```
<template>
  <div class="person">
    <h1 ref="title1">尚硅谷</h1>
    <h2 ref="title2">前端</h2>
    <h3 ref="title3">Vue</h3>
    <input type="text" ref="inpt"> <br><br>
    <button @click="showLog">点我打印内容</button>
  </div>
</template>

<script lang="ts" setup name="Person">
  import {ref} from 'vue'

  let title1 = ref()
  let title2 = ref()
  let title3 = ref()

  function showLog(){
    // 通过id获取元素
    const t1 = document.getElementById('title1')
    // 打印内容
    console.log((t1 as HTMLElement).innerText)
    console.log((<HTMLElement>t1).innerText)
    console.log(t1?.innerText)

    /*****/ 

    // 通过ref获取元素
    console.log(title1.value)
    console.log(title2.value)
    console.log(title3.value)
  }
</script>
```

## 6 TS作用简单总结

在 Vue 中，以 **.ts** 结尾的文件是 **TypeScript** 文件，通常用于编写带有类型检查的逻辑代码。这些文件主要用来定义组件的逻辑部分，包括数据、方法、生命周期钩子等。通过使用 TypeScript，你可以为变量、函数、组件属性等添加类型注解，从而提高代码的健壮性和可维护性。

### 作用概述

- 类型注解**: 为变量、函数参数、返回值等提供明确的类型定义，帮助防止类型错误。
- 增强代码补全和重构能力**: 编辑器可以利用类型信息提供更好的代码提示和重构功能。
- 静态类型检查**: 在编译阶段捕获潜在的错误，减少运行时错误的可能性。

### 例子：下面的人对象和人数组定义

如果name打错成nmae 后面可能会有麻烦 因此需要先写一个ts文件 来定义这些接口的规范

```
<template>
  <div class="person">
    ???
  </div>
```

```

</template>

<script lang="ts" setup name="Person">
    import {type PersonInter,type Persons} from '@/types'

    let person:PersonInter = {id:'asyud7asfd01',name:'张三',age:60}

    let personList:Persons = [
        {id:'asyud7asfd01',name:'张三',age:60},
        {id:'asyud7asfd02',name:'李四',age:18},
        {id:'asyud7asfd03',name:'王五',age:5}
    ]

</script>

<style scoped>
</style>

```

TS文件

```

// 定义一个接口，用于限制person对象的具体属性
export interface PersonInter {
    id:string,
    name:string,
    age:number
}

// 一个自定义类型
// export type Persons = Array<PersonInter>
export type Persons = PersonInter[]

```

## 7 props总结（语法看不懂）

### 1 props 是用于父组件向子组件传递数据

- **单向数据流:** `props` 是用于实现父组件向子组件传递数据的机制。数据流是单向的，即从父组件流向子组件。

### 2 数据流是单向的，而不是双向绑定

- **单向绑定:** `props` 是单向绑定的。数据从父组件传递到子组件，但子组件不能直接修改传入的 `props` 值。如果子组件需要修改数据，通常通过事件通知父组件来改变。

### 3 父组件声明子组件，子组件声明 props

- **父组件:** 在父组件中，通过 `import` 语句引入子组件，然后在模板中使用该子组件，并通过 `props` 将数据传递给它。
- **子组件:** 在子组件中，需要使用 `defineProps` 函数（或 Vue 2 中的 `props` 选项）来声明将接收哪些 `props` 以及它们的类型。

## 完整的总结

- **props 是 Vue 中父组件向子组件传递数据的机制。**通过 `props`，父组件可以将数据传递给子组件，使得子组件能够根据这些数据进行渲染或执行逻辑。
- **数据流是单向的：**`props` 的数据从父组件流向子组件，而子组件不能直接修改 `props` 的值。需要修改时，子组件通常会通过事件通知父组件，由父组件来更新数据。
- **父组件中声明子组件：**在父组件中，你需要引入子组件并在模板中使用它，同时通过 `props` 传递数据。

```
<template>
  <!-- 父组件中使用子组件，并通过 props 传递数据 -->
  <abc :count="number" />
</template>

<script setup>
import abc from './ChildComponent.vue';

const number = 42; // 父组件中的数据
</script>
```

- **子组件中声明 props：**子组件通过 `defineProps` 函数来声明它将接收哪些 `props`，并指定这些 `props` 的类型

```
<template>
  <div>
    <p>传递过来的数字是：{{ count }}</p>
  </div>
</template>

<script setup>
// 子组件中声明 props
const props = defineProps({
  count: Number // 这里声明 count 是一个数字类型的 prop
});
</script>
```

接下来灵活一点 上传一个人对象  
父组件：

```
<template>
  <!-- 传递一个表示人的对象 -->
  <PersonComponent :person="personData" />
</template>

<script setup>
import PersonComponent from './PersonComponent.vue';

// 定义一个表示人的对象
const personData = {
  name: '张三',
  age: 30,
  address: {
    city: '北京',
    street: '长安街'
  },
  hobbies: ['阅读', '跑步']
};
```

```
</script>
```

子组件

```
<template>
  <div>
    <p>姓名: {{ person.name }}</p>
    <p>年龄: {{ person.age }}</p>
    <p>城市: {{ person.address.city }}</p>
    <p>街道: {{ person.address.street }}</p>
    <p>爱好: {{ person.hobbies.join(', ') }}</p>
  </div>
</template>

<script setup>
const props = defineProps({
  person: {
    type: Object,
    required: true
  }
});
```

## 8. 自定义Hook函数

在 Vue.js 中，**hook**（钩子）通常指的是**生命周期钩子**。这些钩子函数是 Vue 组件在不同生命周期阶段自动调用的一些函数，它们让开发者能够在组件的特定时刻执行代码。

- 什么是 **hook**？——本质是一个函数，把 **setup** 函数中使用的 **Composition API** 进行了封装，类似于 **vue2.x** 中的 **mixin**。
- 自定义 **hook** 的优势：复用代码，让 **setup** 中的逻辑更清楚易懂。
- vue3** 的生命周期

创建阶段：**setup**  
挂载阶段：**onBeforeMount**、**onMounted**  
更新阶段：**onBeforeUpdate**、**onUpdated**  
卸载阶段：**onBeforeUnmount**、**onUnmounted**

- 常用的钩子：**onMounted**（挂载完毕）、**onUpdated**（更新完毕）、**onBeforeUnmount**（卸载之前）

例如，让一些组件元素一开始就有个值，或者让一个展示窗口一开始就有一张图片放进去 就可以调用 **onMounted**（挂载完毕）

```
import {reactive, onMounted} from 'vue'
import axios from 'axios'

export default function () {
  // 数据
  let dogList = reactive([
    'https://images.dog.ceo/breeds/pembroke/n02113023_4373.jpg'
  ])
  // 方法
  async function getDog() {
```

```

try {
  let result = await
axios.get('https://dog.ceo/api/breed/pembroke/images/random')
  dogList.push(result.data.message)
} catch (error) {
  alert(error)
}
}

async function deleteDog(){
  try {
    dogList.pop()
  } catch (error) {
    alert(error)
  }
}

// 钩子
onMounted(()=>{
  getDog()
  getDog()
})
// 向外部提供东西
return {dogList,getDog,deleteDog}
}

```

## 9, route路由

### 9.1 route注意点

1. 路由组件通常存放在 `pages` 或 `views` 文件夹，一般组件通常存放在 `components` 文件夹。
2. 通过点击导航，视觉效果上“消失”了的路由组件，默认是被卸载掉的，需要的时候再去挂载。（用钩子函数可以验证）

### 9.2 路由器的工作模式

#### 1. `history` 模式

优点：URL 更加美观，不带有 #，更接近传统的网站 URL。

缺点：后期项目上线，需要服务端配合处理路径问题，否则刷新会有 404 错误。

```

const router = createRouter({
  history:createWebHistory(), //history模式
  /**
})

```

#### 2. `hash` 模式

优点：兼容性更好，因为不需要服务器端处理路径。

缺点：URL 带有 # 不太美观，且在 SEO 优化方面相对较差。

```
const router = createRouter({
  history:createWebHashHistory(), //hash模式
  /**
  */
})
```

### 9.3 路由传参

先说结论 这个功能语法复杂 且仅适用于传递简单的参数 所以了解概念即可

功能：让不同的路由之间，以URL的形式传递信息

分为query和param两种 语法有很大区别

重点：这个功能很难用简单的例子来形象复习 建议有需求的时候直接回去看视频

①在news.vue中，在连接中添加了参数信息

```
<template>
  <div class="news">
    <!-- 导航区 -->
    <ul>
      <li v-for="news in newsList" :key="news.id">
        <RouterLink
          :to="{
            path:'/news/detail',
            query:{
              id:news.id,
              title:news.title,
              content:news.content
            }
          }"
        >
          {{news.title}}
        </RouterLink>

      </li>
    </ul>
    <!-- 展示区 -->
    <div class="news-content">
      <RouterView></RouterView>
    </div>
  </div>
</template>

<script setup lang="ts" name="News">
  import {reactive} from 'vue'
  import {RouterView,RouterLink} from 'vue-router'

  const newsList = reactive([
    {id:'asfdtrfay01',title:'很好的抗癌食物',content:'西兰花'},
    {id:'asfdtrfay02',title:'如何一夜暴富',content:'学IT'},
    {id:'asfdtrfay03',title:'震惊，万万没想到',content:'明天是周一'},
    {id:'asfdtrfay04',title:'好消息！好消息！',content:'快过年了'}
  ])
</script>

<style scoped>
</style>
```

②在detail详情页面声明和接受了参数信息 并且使用

```

<template>
  <ul class="news-list">
    <li>编号: {{ query.id }}</li>
    <li>标题: {{ query.title }}</li>
    <li>内容: {{ query.content }}</li>
  </ul>
</template>

<script setup lang="ts" name="About">
  import {toRefs} from 'vue'
  import {useRoute} from 'vue-router'
  let route = useRoute()
  let {query} = toRefs(route)

</script>

<style scoped>
</style>

```

## 9.4 路由中的props

作用：让路由组件更方便的收到参数（可以将路由参数作为 **props** 传给组件）

```

{
  name:'xiang',
  path:'detail/:id/:title/:content',
  component:Detail,
}

// props的对象写法，作用：把对象中的每一组key-value作为props传给Detail组件
props:{a:1,b:2,c:3},

// props的布尔值写法，作用：把收到了每一组params参数，作为props传给Detail组件
props:true

// props的函数写法，作用：把返回的对象中每一组key-value作为props传给Detail组件
props(route){
  return route.query
}
}

```

## 9.5 ! ! ! 编程式路由导航! ! !

非常有用！！

使用案例：

1, 登录成功跳转 2, 固定时间跳转

因为 这里的条件判断通过组件是无法完成的 需要函数

路由组件的两个重要的属性：**\$route** 和 **\$router** 变成了两个 **hooks**

```

import {useRoute,useRouter} from 'vue-router'

const route = useRoute()
const router = useRouter()

onMounted(()=>{
  setTimeout(()=>{
    router.push('/news')
  },3000)
})

```

补充一个重定向

1. 作用： 将特定的路径，重新定向到已有路由。

2. 具体编码：

```
{  
  path: '/', //这个路径走不通  
  redirect: '/about'  
}
```

## 10 Pinia 状态管理工具

共享的数据交给pinia 集中式状态管理 方便不同的组件共享

### 10.1，使用方法

1. **Store** 是一个保存：**状态、业务逻辑** 的实体，每个组件都可以**读取、写入**它。
2. 它有三个概念：**state、getter、action**，相当于组件中的：**data、computed 和 methods**。
3. 具体编码：**src/store/count.ts**

```
// 引入defineStore用于创建store  
import {defineStore} from 'pinia'  
  
// 定义并暴露一个store  
export const useCountStore = defineStore('count', {  
  // 动作  
  actions:{},  
  // 状态  
  state(){  
    return {  
      sum:6  
    }  
  },  
  // 计算  
  getters:{}  
})
```

4. 具体编码：**src/store/talk.ts**

```
// 引入defineStore用于创建store  
import {defineStore} from 'pinia'  
  
// 定义并暴露一个store  
export const useTalkStore = defineStore('talk', {  
  // 动作  
  actions:{},  
  // 状态  
  state(){  
    return {  
      talkList:[  
        {id:'yuysada01',content:'你今天有点怪，哪里怪？怪好看的！'},  
        {id:'yuysada02',content:'草莓、蓝莓、蔓越莓，你想我了没？'},  
        {id:'yuysada03',content:'心里给你留了一块地，我的死心塌地'}  
      ]  
    }  
  },  
  // 计算  
  getters:{}  
})
```

```
}
```

## 5. 组件中使用 state 中的数据

```
<template>
  <h2>当前求和为: {{ sumStore.sum }}</h2>
</template>

<script setup lang="ts" name="Count">
  // 引入对应的useXXXXStore
  import {useSumStore} from '@/store/sum'

  // 调用useXXXXStore得到对应的store
  const sumStore = useSumStore()
</script>
```

```
<template>
  <ul>
    <li v-for="talk in talkstore.talkList" :key="talk.id">
      {{ talk.content }}
    </li>
  </ul>
</template>

<script setup lang="ts" name="Count">
  import axios from 'axios'
  import {useTalkStore} from '@/store/talk'

  const talkStore = useTalkStore()
</script>
```

## 10.2 如何修改pinia里的数据 (不熟悉没关系, 直接看肯定看得懂 )

### 1. 第一种修改方式, 直接修改或者批量修改

```
<script setup lang="ts" name="Count">
  import { ref,reactive } from "vue";
  // 引入useCountStore
  import {useCountStore} from '@/store/count'
  // 使用useCountStore, 得到一个专门保存count相关的store
  const countStore = useCountStore()

  // 数据
  let n = ref(1) // 用户选择的数字
  // 方法
  function add(){
    // 第一种修改方式
    countStore.sum += 1

    // 第二种修改方式
    countStore.$patch({
      sum:888,
      school:'尚硅谷',
      address:'北京'
    })

    // 第三种修改方式
    //countStore.increment(n.value)
  }
</script>
```

```
}
```

2. 第二种修改方式：借助 **action** 修改（**action**中可以编写一些业务逻辑）

```
import { defineStore } from 'pinia'

export const useCountStore = defineStore('count', {
    //*****
    actions: {
        //加
        increment(value:number) {
            if (this.sum < 10) {
                //操作countStore中的sum
                this.sum += value
            }
        },
        //减
        decrement(value:number){
            if(this.sum > 1){
                this.sum -= value
            }
        }
    },
    //*****
})
```

3. 组件中调用 **action** 即可

```
// 使用countStore
const countStore = useCountStore()

// 调用对应action
countStore.increment(n.value)
```

### 10.3, **getter** (和**action**修改数据差不多，只是用途不同)

1. 概念：当 **state** 中的数据，需要经过处理后再使用时，可以使用 **getters** 配置。
2. 追加 **getters** 配置。

```
// 引入defineStore用于创建store
import {defineStore} from 'pinia'

// 定义并暴露一个store
export const useCountStore = defineStore('count',{
    // 动作
    actions:{
        //*****
    },
    // 状态
    state(){
        return {
            sum:1,
            school:'atguigu'
        }
    }
})
```

```

},
// 计算
getters:{
    bigSum:(state):number => state.sum *10,
    upperschool():string{
        return this.school.toUpperCase()
    }
}
)

```

3. 组件中读取数据：（就是把函数也一起返回了）

```

const {increment,decrement} = countStore
let {sum,school,bigsum,upperschool} = storeToRefs(countStore)

```

**Getters**: 用于派生状态，提供计算属性，通常不会直接修改状态。适合用来基于现有状态生成新的数据或进行状态的计算。

**Actions**: 用于封装业务逻辑，可以直接修改状态或进行异步操作。适合执行多步骤逻辑、异步请求或复杂状态更改

## 10.4 subscribe监听（就是watch）

在lovetalk.vue中添加监听功能

这里的 `(mutate, state)` 是自动回调的 所以没有这两个参数，但是默认这么写  
然后通过这两个参数来监听状态 已经封装起来了 理解这一点就能看懂了

```

<template>
<div class="talk">
    <button @click="getLoveTalk">获取一句土味情话</button>
    <ul>
        <li v-for="talk in talkList" :key="talk.id">{{talk.title}}</li>
    </ul>
</div>
</template>

<script setup lang="ts" name="LoveTalk">
import {useTalkStore} from '@/store/loveTalk'
import { storeToRefs } from "pinia";

const talkStore = useTalkStore()
const {talkList} = storeToRefs(talkStore)
//添加的监听功能
talkStore.$subscribe((mutate,state)=>{
    console.log('talkStore里面保存的数据发生了变化',mutate,state)
    localStorage.setItem('talkList',JSON.stringify(state.talkList))
})

// 方法
function getLoveTalk(){
    talkStore.getATalk()
}
</script>

<style scoped>
</style>

```

## 10.5 组合式API (Vue3特性)

这里复习vue3的内容 vue2的选项式和vue3的组合式

### 1 选项式 API (Vue 2 的风格)

这是 Vue 2 的写法，使用 `data`、`methods`、`computed` 等来组织代码：

```
export default {
  data() {
    return {
      count: 0,
    };
  },
  methods: {
    increment() {
      this.count++;
    },
  },
};
```

在这个例子中：

- `data` 用来定义组件的状态 (`count`)。
- `methods` 用来定义组件的行为 (`increment` 方法)。

### 2 组合式 API (Vue 3 的新风格)

这是 Vue 3 的写法，使用 `setup` 函数将状态和逻辑组合在一起：

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

export default {
  setup() {
    const count = ref(0);

    function increment() {
      count.value++;
    }

    return { count, increment };
  },
};
```

组合式 API 中，`setup` 函数是唯一的入口，`ref` 用来创建响应式的状态，逻辑和状态都在 `setup` 中定义并返回。

### 3 pinia的组合式写法

选项式：分了action和state 分别对应方法以及数据

```
import {defineStore} from 'pinia'
import axios from 'axios'
import {nanoid} from 'nanoid'
```

```

export const useTalkStore = defineStore('talk', {
  actions: {
    async getATalk(){
      // 发请求，下面这行的写法是：连续解构赋值+重命名
      let {data:{content:title}} = await
      axios.get('https://api.uomg.com/api/rand.qinghua?format=json')
      // 把请求回来的字符串，包装成一个对象
      let obj = {id:nanoid(),title}
      // 放到数组中
      this.talkList.unshift(obj)
    }
  },
  // 真正存储数据的地方
  state(){
    return {
      talkList:JSON.parse(localStorage.getItem('talkList') as string) || []
    }
  }
})

```

组合式：取消了这两个直接写到一起 然后返回

```

import {defineStore} from 'pinia'
import axios from 'axios'
import {nanoid} from 'nanoid'
import {reactive} from 'vue'

export const useTalkStore = defineStore('talk', ()=>{
  // talkList就是state
  const talkList = reactive(
    JSON.parse(localStorage.getItem('talkList') as string) || []
  )

  // getATalk函数相当于action
  async function getATalk(){
    // 发请求，下面这行的写法是：连续解构赋值+重命名
    let {data:{content:title}} = await
    axios.get('https://api.uomg.com/api/rand.qinghua?format=json')
    // 把请求回来的字符串，包装成一个对象
    let obj = {id:nanoid(),title}
    // 放到数组中
    talkList.unshift(obj)
  }
  return {talkList,getATalk}
})

```

## 11 组件通信（总览）

这里为了防止概念混淆 先把目前所有可以实现组件通信的方式做一个总结  
在实际开发中，最常用且最值得掌握的三种组件间信息传递方式是：

## 1. Props and Emit

- **最常用场景**: 父子组件之间的数据传递。
- **为什么要掌握**: 这是 Vue 中最基本且核心的组件间通信方式。通过 `props`, 父组件可以将数据传递给子组件；通过 `emit`, 子组件可以向父组件发送事件。这种单向数据流是构建 Vue 组件树的基础。
- **学习难度**: 低，易于理解和使用。

## 2. Pinia (or Vuex for older projects)

- **最常用场景**: 全局状态管理，多组件之间共享状态，尤其是大型应用程序。
- **为什么要掌握**: Pinia 是 Vue 官方推荐的状态管理库，适合在复杂应用中管理全局状态，解决了组件之间数据共享和通信的问题。了解和掌握 Pinia 能让你更好地处理复杂的数据流和状态管理。
- **学习难度**: 中等，涉及到状态管理的概念和模式。

## 3. Vue Router

- **最常用场景**: 页面级别的组件之间通过 URL 进行数据传递和导航。
- **为什么要掌握**: 在开发多页面或单页应用 (SPA) 时，Vue Router 是必备的工具。通过路由传递参数，可以让你在不同页面组件之间传递数据，非常适合在导航过程中传递信息，如在详情页中显示列表页的数据。
- **学习难度**: 中等，涉及到路由配置、导航守卫等概念。

## 总结：

- **Props and Emit** 是基础，任何 Vue 开发者都需要掌握。
- **Pinia** 是现代 Vue 应用中管理复杂状态的利器，尤其是在需要跨组件共享状态时。
- **Vue Router** 是实现页面间导航和参数传递的关键工具，在多页面应用中非常重要。

# 12 组件通信props

## 12.1. props本体

概述：`props` 是使用频率最高的一种通信方式，常用与：**父 ↔ 子**。

- 若**父传子**：属性值是**非函数**，很好理解 就是父亲给儿子玩具 儿子接受
- 若**子传父**：属性值是**函数**。这个难理解，需要：
  - 1: 父亲先给儿子传一个函数 `getToy`
  - 2: 儿子通过这个函数`getToy`来实现给父亲传玩具的功能

父组件：

```
<template>
  <div class="father">
    <h3>父组件, </h3>
    <h4>我的车: {{ car }}</h4>
    <h4>儿子给的玩具: {{ toy }}</h4>
    <Child :car="car" :getToy="getToy"/>
  </div>
</template>

<script setup lang="ts" name="Father">
  import Child from './Child.vue'
  import { ref } from "vue";
  // 数据
  const car = ref('奔驰')
```

```

const toy = ref()
// 方法
function getToy(value:string){
    toy.value = value
}
</script>

```

子组件

```

<template>
<div class="child">
    <h3>子组件</h3>
    <h4>我的玩具: {{ toy }}</h4>
    <h4>父给我的车: {{ car }}</h4>
    <button @click="getToy(toy)">玩具给父亲</button>
</div>
</template>

<script setup lang="ts" name="Child">
import { ref } from "vue";
const toy = ref('奥特曼')

defineProps(['car','getToy'])
</script>

```

## 12.2. 自定义事件

1. 概述：自定义事件常用于：**子 => 父**。
2. 注意区分好：原生事件、自定义事件。
  - 原生事件：
    - 事件名是特定的 (`click`、`mouseenter`等等)
    - 事件对象 `$event`：是包含事件相关信息的对象 (`pageX`、`pageY`、`target`、`keyCode`)
  - 自定义事件：
    - 事件名是任意名称
    - **事件对象 `$event`：是调用 `emit` 时所提供的数据，可以是任意类型！！！**
3. 示例：

```

<!--在父组件中，给子组件绑定自定义事件：-->
<Child @send-toy="myEvent"/>
</div>
</template>

<script setup lang="ts" name="Father">
import Child from './Child.vue'
import { ref } from "vue";
// 数据
let toy = ref('')
// 用于保存传递过来的玩具
function myEvent(value:string){
    console.log('saveToy',value)
    toy.value = value
}

```

```

//子组件中，触发事件：
<template>

```

```

<div class="child">
  <h3>子组件</h3>
  <h4>玩具: {{ toy }}</h4>
  <button @click="emit('myEvent',toy)">测试</button>
</div>
</template>

<script setup lang="ts" name="Child">
  import { ref } from "vue";
  // 数据
  let toy = ref('奥特曼')
  // 声明事件
  const emit = defineEmits(['myEvent'])
</script>

```

### 12.3. 【mitt】

概述：与消息订阅与发布（pubsub）功能类似，可以实现任意组件间通信。

安装mitt

```
npm i mitt
```

新建文件：`src\utils\emitter.ts`

假设我们有一个简单的购物车应用，当用户点击“添加到购物车”按钮时，页面上购物车的图标需要更新以反映当前购物车中商品的数量。

简单教程

```

// 引入mitt
import mitt from 'mitt'

// 调用mitt得到emitter, emitter能：绑定事件、触发事件
const emitter = mitt()

// 绑定事件
emitter.on('test1', ()=>{
  console.log('test1被调用了')
})
emitter.on('test2', ()=>{
  console.log('test2被调用了')
})

// 触发事件
setInterval(() => {
  emitter.emit('test1')
  emitter.emit('test2')
}, 1000);

setTimeout(() => {
  emitter.off('test1')
  emitter.off('test2')
  emitter.all.clear()
}, 3000);

```

**核心：这里的事件名称是自定义的 也就是在绑定的时候自己取的名字 这里就是定义的地点**

以上面为例，

`emitter.on('test1', ()=>{``console.log('test1被调用了')})``实际上是事件的定义+绑定  
后续需要调用的时候，再用`emitter.emit('test1')`调用

## 1. 设置事件系统

首先，我们创建一个`mitt`实例来管理事件：

```
import mitt from 'mitt';

const emitter = mitt();
export default emitter;
```

## 2. 在商品页面中发布事件

在商品页面中，当用户点击“添加到购物车”按钮时，我们会发布一个事件，通知系统购物车中的商品数量发生了变化：

```
import emitter from './emitter';

function addToCart(product) {
    // 这里是添加商品到购物车的逻辑
    // ...

    // 发布一个事件，通知购物车数量发生了变化
    emitter.emit('cartUpdated', { product, count: 1 });
}

document.getElementById('add-to-cart-button').addEventListener('click', () => {
    addToCart({ id: 1, name: 'Example Product' });
});
```

## 3. 在购物车图标组件中订阅事件

接下来，我们在购物车图标的组件中订阅这个事件，以便在事件发生时更新购物车的显示数量：

```
import emitter from './emitter';

function updateCartIcon(data) {
    const cartIcon = document.getElementById('cart-icon');
    const currentCount = parseInt(cartIcon.innerText, 10) || 0;
    cartIcon.innerText = currentCount + data.count;
}

// 订阅 'cartUpdated' 事件
emitter.on('cartUpdated', updateCartIcon);
```

## 4. 完整的流程

- 用户点击商品页面的“添加到购物车”按钮。
- `addToCart` 函数执行并通过`emitter.emit`发送`cartUpdated`事件，同时传递商品信息和数量。
- 购物车图标组件接收到`cartUpdated`事件，`updateCartIcon`函数被调用，并更新购物车图标的显示数量。

## 12.4 v-model

实在看不太懂 给出gpt的解释 反正实际用的也不多  
实现 父↔子 之间相互通信。

### 1. 父组件的初始状态

在父组件 `Father` 中，你定义了两个 `ref` 变量：`username` 和 `password`，用于存储输入的用户名和密码。

```
let username = ref('zhansgan');
let password = ref('123456');
```

你还使用了 `v-model` 语法糖，将这两个变量绑定到子组件 `AtguiguInput` 上：

```
vue
复制代码
<AtguiguInput v-model:ming="username" v-model:mima="password"/>
```

### 2. 子组件的接收与显示

在子组件 `AtguiguInput` 中，`ming` 和 `mima` 是通过 `defineProps` 接收的父组件传递过来的数据。

```
defineProps(['ming', 'mima']);
```

子组件将 `ming` 和 `mima` 绑定到两个输入框的 `value` 属性上，以便显示传递过来的 `username` 和 `password`。

```
<input
  type="text"
  :value="ming"
  @input="emit('update:ming', (<HTMLInputElement>$event.target).value)"
>
<input
  type="text"
  :value="mima"
  @input="emit('update:mima', (<HTMLInputElement>$event.target).value)"
>
```

### 3. 子组件中用户输入

当用户在子组件的第一个输入框中输入新内容时，`@input` 事件被触发：

```
@input="emit('update:ming', (<HTMLInputElement>$event.target).value)"
```

这行代码做了以下几件事：

1. **获取输入的值：** `(<HTMLInputElement>$event.target).value` 获取当前输入框中的值。
2. **触发事件：** 调用 `emit('update:ming', newValue)`，触发一个名为 `update:ming` 的事件，并将获取的值作为参数传递。

### 4. 父组件接收事件

当子组件触发 `update:ming` 事件时，父组件会自动接收到这个事件并更新对应的变量 `username`。这是因为你在父组件中使用了 `v-model:ming`：

```
<AtguiguInput v-model:ming="username" v-model:mima="password"/>
```

Vue 自动为你处理了事件绑定，相当于做了以下事情：

```
<AtguiguInput  
  :ming="username"  
  @update:ming="username = $event"  
  :mima="password"  
  @update:mima="password = $event"  
/>
```

当 `emit('update:ming', newValue)` 被触发时，父组件的 `username` 自动被更新为 `newValue`。

## 5. 父组件显示更新

随着 `username` 和 `password` 在父组件中被更新，它们的最新值会在父组件的模板中显示出来：

```
<h4>{{ username }}</h4>  
<h4>{{ password }}</h4>
```

这意味着当用户在子组件中输入时，父组件会实时显示更新后的值。

## 6. 总结

- **父组件初始化数据**: `username` 和 `password` 被定义并传递到子组件中。
- **子组件接收并显示数据**: 子组件通过 `ming` 和 `mima` 接收并显示数据。
- **用户输入触发事件**: 当用户输入时，子组件触发 `update:ming` 和 `update:mima` 事件，带有新输入的值。
- **父组件更新并显示**: 父组件自动接收事件并更新 `username` 和 `password`，从而在页面上显示新的输入值。

通过 `v-model` 的双向绑定机制，父子组件之间的数据传递和同步变得简单且高效。

## 12.5 attrs

这里GPT写更明白

1. 概述：`$attrs` 用于实现**当前组件的父组件**，向**当前组件的子组件**通信（祖→孙）。
2. 具体说明：`$attrs` 是一个对象，包含所有父组件传入的标签属性。

### 1. 传递链条的概述

在这个例子中，属性的传递链条如下：

- **祖父组件 (Father)** → **父组件 (Child)** → **孙组件 (Grandchild)**

具体来说：

1. **祖父组件 (Father)**：将 `a`、`b`、`c`、`d`、`x`、`y` 这些属性，以及 `updateA` 方法，传递给 `Child` 组件。
2. **父组件 (Child)**：`Child` 组件接收到这些属性后，并没有直接使用它们，而是通过 `$attrs` 将这些属性传递给 `Grandchild` 组件。
3. **孙组件 (Grandchild)**：`Grandchild` 通过 `defineProps` 来接收这些属性，并在模板中使用它们。
4. **原因**：父亲向下传递的是数据，而孙子的按钮是通过事件来改变父亲 不是一回事

### 2. 为什么是父向孙传递？

你提到的“父向孙传递”实际上是指 `Child` 组件并没有处理这些属性，而是通过 `$attrs` 将它们传递给了 `Grandchild`。从整体流程来看：

- **父组件 (Child)** 通过 `$attrs` 将它自己收到的属性再传递给它的子组件 (`Grandchild`)。这是一种简化属性传递的方式，让中间的 `Child` 组件不需要关心这些属性的具体内容，只需要负责将它们“转发”给下一层的组件。

- 最终的效果是，`GrandChild` 组件实际上收到了这些属性，并且可以在它的模板中使用它们。

### 3. 总结与流程

1. 祖父组件 (`Father`)：定义并传递了属性和方法。
2. 父组件 (`Child`)：接收这些属性和方法，但并未显式声明它们，而是通过 `$attrs` 转发给 `GrandChild`。
3. 孙组件 (`Grandchild`)：使用 `defineProps` 明确声明接收的属性，并在模板中使用它们。

这种传递方式可以让组件保持高度的解耦性和可复用性，尤其是当你不希望在中间组件中手动处理所有属性时，使用 `$attrs` 可以简化代码结构。

### 4. 实际工作流程

1. 属性传递：`Father` 组件传递属性到 `Child` 组件。
2. 属性转发：`Child` 组件使用 `$attrs` 将属性转发到 `Grandchild` 组件。
3. 属性接收：`Grandchild` 组件通过 `defineProps` 接收属性并在模板中显示。

父组件：

```
<template>
  <div class="father">
    <h3>父组件</h3>
    <Child :a="a" :b="b" :c="c" :d="d" v-bind="{x:100,y:200}"
:updateA="updateA"/>
  </div>
</template>

<script setup lang="ts" name="Father">
  import Child from './child.vue'
  import { ref } from "vue";
  let a = ref(1)
  let b = ref(2)
  let c = ref(3)
  let d = ref(4)

  function updateA(value){
    a.value = value
  }
</script>
```

子组件：

```
<template>
  <div class="child">
    <h3>子组件</h3>
    <GrandChild v-bind="$attrs"/>
  </div>
</template>

<script setup lang="ts" name="Child">
  import GrandChild from './GrandChild.vue'
</script>
```

孙组件：

```
<template>
  <div class="grand-child">
```

```

<h3>孙组件</h3>
<h4>a: {{ a }}</h4>
<h4>b: {{ b }}</h4>
<h4>c: {{ c }}</h4>
<h4>d: {{ d }}</h4>
<h4>x: {{ x }}</h4>
<h4>y: {{ y }}</h4>
<button @click="updateA(666)">点我更新A</button>
</div>
</template>

<script setup lang="ts" name="GrandChild">
  defineProps(['a','b','c','d','x','y','updateA'])
</script>

```

## 12.6 \$refs和\$parent

1. 概述:

- \$refs 用于 : 父→子。
- \$parent 用于: 子→父。

2. 原理如下:

属性	说明
\$refs	值为对象, 包含所有被 ref 属性标识的 DOM 元素或组件实例。
\$parent	值为对象, 当前组件的父组件实例对象

### 1. \$refs 简单形象的解释与示例

**解释:** \$refs 是 Vue 3 中用来直接访问 DOM 元素或子组件的引用。通过在模板中使用 ref, 你可以为某个元素或组件创建一个引用, 然后在脚本部分通过 const 访问它。

**示例:**

假设我们有一个按钮, 点击它时希望输入框自动获取焦点。

```

<template>
  <div>
    <!-- 给 input 元素一个引用名 myInput -->
    <input ref="myInput" type="text" placeholder="点按钮后我会自动聚焦">
    <button @click="focusInput">聚焦输入框</button>
  </div>
</template>

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

// 定义一个引用来存储 input 元素的 DOM 对象
const myInput = ref(null);

function focusInput() {
  // 通过 myInput.value 访问 input 元素并调用其 focus 方法
  myInput.value.focus();
}
</script>

```

- 解释：在这个例子中，`ref="myInput"` 为输入框创建了一个引用。通过 `myInput.value` 可以访问到这个输入框元素，并调用 `focus()` 方法让它获得焦点。

## 1 对于实际例子中的，具有多个ref引用时：

```

<template>
  <div class="father">
    <h3>父组件</h3>
    <h4>房产: {{ house }}</h4>
    <button @click="changeToy">修改child1的玩具</button>
    <button @click="changeComputer">修改child2的电脑</button>
    <button @click="getAllChild($refs)">让所有孩子的书变多</button>
    <child1 ref="c1"/>
    <child2 ref="c2"/>
  </div>
</template>

<script setup lang="ts" name="Father">
  import Child1 from './child1.vue'
  import Child2 from './child2.vue'
  import { ref, reactive } from "vue";
  let c1 = ref()
  let c2 = ref()

  // 数据
  let house = ref(4)
  // 方法
  function changeToy(){
    c1.value.toy = '小猪佩奇'
  }
  function changeComputer(){
    c2.value.computer = '华为'
  }
  function getAllChild(refs:{[key:string]:any}){
    console.log(refs)
    for (let key in refs){
      refs[key].book += 3
    }
  }
  // 向外部提供数据
  defineExpose({house})
</script>

<style scoped>
</style>

```

在这个例子中，`ref="c1"` 和 `ref="c2"`：你为 `Child1` 和 `Child2` 组件添加了 `ref`。这些 `ref` 会将子组件的实例保存到 `$refs.c1` 和 `$refs.c2` 中，分别对应 `Child1` 和 `Child2` 的组件实例。`$refs: $refs` 是 Vue 自动生成的对象，包含了所有被 `ref` 标记的子组件或 DOM 元素。在你的例子中，`$refs` 会包含 `c1` 和 `c2`，即 `child1` 和 `child2` 的实例。

2 在父组件中，你定义了一些方法来操作子组件的属性或调用它们的方法：

```
function changeToy() {
  $refs.c1.toy = '新的玩具'; // 修改 child1 组件的 toy 属性
}

function changeComputer() {
  $refs.c2.computer = '新的电脑'; // 修改 child2 组件的 computer 属性
}

function getAllChild(refs) {
  for (let key in refs) {
    refs[key].book += 3; // 遍历所有子组件，并增加它们的 book 属性值
  }
}
```

- `changeToy`: 这个方法通过 `$refs.c1` 访问 `child1` 组件的实例，然后修改它的 `toy` 属性。
- `changeComputer`: 这个方法通过 `$refs.c2` 访问 `child2` 组件的实例，然后修改它的 `computer` 属性。
- `getAllChild`: 这个方法接收 `$refs` 作为参数，遍历其中的每个子组件实例，并增加它们的 `book` 属性值。

## 2. \$parent 简单形象的解释与示例

**解释**: 在 Vue 3 中，`$parent` 用于让子组件访问父组件的实例。它可以让子组件访问父组件的属性或调用父组件的方法。

不过，使用 `$parent` 会增加组件之间的耦合度，因此通常推荐使用 `props` 和 `emit` 事件来进行父子组件通信。

所以不学了

## 12.7 provide inject

1. 概述：实现**祖孙组件**直接通信 不涉及到其他的中间组件

2. 具体使用：

- 在祖先组件中通过 `provide` 配置向后代组件提供数据
- 在后代组件中通过 `inject` 配置来声明接收数据

**attrs**: 用来将父组件传递但未被接收的属性传递到子组件中。它通常用于直接的父子组件通信。

**provide/inject**: 适合于更深层次的组件通信，特别是在需要跨越多个组件层级传递数据或方法时。它可以避免通过中间组件逐层传递 `props`，让代码更简洁。

1. 具体编码：

【第一步】父组件中，使用 `provide` 提供数据

```
<template>
<div class="father">
  <h3>父组件</h3>
  <h4>资产: {{ money }}</h4>
  <h4>汽车: {{ car }}</h4>
  <button @click="money += 1">资产+1</button>
  <button @click="car.price += 1">汽车价格+1</button>
  <child/>
```

```
</div>
</template>

<script setup lang="ts" name="Father">
  import Child from './Child.vue'
  import { ref,reactive,provide } from "vue";
  // 数据
  let money = ref(100)
  let car = reactive({
    brand:'奔驰',
    price:100
  })
  // 用于更新money的方法
  function updateMoney(value:number){
    money.value += value
  }
  // 提供数据
  provide('moneyContext',{money,updateMoney})
  provide('car',car)
</script>
```

注意：子组件中不用编写任何东西，是不受到任何打扰的

【第二步】孙组件中使用 `inject` 配置项接受数据。

```
<template>
  <div class="grand-child">
    <h3>我是孙组件</h3>
    <h4>资产: {{ money }}</h4>
    <h4>汽车: {{ car }}</h4>
    <button @click="updateMoney(6)">点我</button>
  </div>
</template>

<script setup lang="ts" name="Grandchild">
  import { inject } from 'vue';
  // 注入数据
  let {money,updateMoney} = inject('moneyContext',{money:0,updateMoney:(x:number)=>{}})
  let car = inject('car')
</script>
```

## 13 插槽

相当有意思且实用的功能，但原版笔记过于晦涩，此处自行总结

### 13.1 插槽是什么

```

<template>
  <div class="father">
    <h3>父组件</h3>
    <div class="content">
      <Category title="热门游戏列表">
        <span>你好</span>
      </Category>
      <!-- <Category title="今日美食城市"/> -->
      <!-- <Category title="今日影视推荐"/> -->
    </div>
  </div>
</template>

1  <template>
2  | <div class="category">
3  |   <h2>{{title}}</h2>
4  |   0 references
5  |   <slot></slot>
6  | </div>
7
8  <script setup lang="ts" name="Category">
9  |   defineProps(['title'])
10 </script>
11
12 <style scoped> ...

```

右图作为子组件，本来是只需要反括号的，但是这样 **很难父组件中引入更复杂的数据形式**

**解决办法：**写成双括号 然后再中间添加内容

比如这是我想添加的更复杂的内容 是一个自定义的表格

```

<Category>
  <template v-slot="params">
    <ul>
      <li v-for="y in params.youxi" :key="y.id">
        {{ y.name }}
      </li>
    </ul>
  </template>
</Category>

```

**问题：**系统不知道该添加到哪里 因此不会添加

**再次解决：**通过在右边的子组件中添加这个标识 就会插入到slot所在的位置  
这里的默认内容插入了之后会被顶掉 变成我写的“你好”

<slot>默认插入位置</slot>

这样，我就可以插入任何我想要的数据了，表格图片超链接甚至视频等等

## 13.2 具名插槽

有了上面的功能 已经可以实现复杂数据的传递了

**问题：**这样还是不够灵活 我希望子组件能够有判断具体呈现哪一个数据的功能

**解决方法：**给两边都取名字

```

父组件中:
<Category title="今日热门游戏">
  <template v-slot:s1>
    <ul>
      <li v-for="g in games" :key="g.id">{{ g.name }}</li>
    </ul>
  </template>
  <template #s2>
    <a href="">更多</a>
  </template>
</Category>

子组件中:
<template>
  <div class="item">
    <h3>{{ title }}</h3>

```

```
<slot name="s1"></slot>
<slot name="s2"></slot>
</div>
</template>
```

上面不同的template还取了不同名字 方便插入不同的插槽来控制排布

### 13.3 作用域插槽