Vue组件及组件化开发

在现行的MVVM的框架里面（vue,react,angular），它们都是支持组件化开发的，组件化开发也叫`virtual DOM`开发

存在的问题

我们有时候在进行页面开发的时候，发现页面上面某些地方很个似，这个时候怎么办呢？如下

```
<!DOCTYPE html>
<html lang="zh">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>vue组件及组件化开发</title>
    <style>
        .list-group{
            margin: 0;
            padding: 0;
            list-style-type: none;
        }
        .list-group>li{
            min-height: 45px;
            display: flex;
            align-items: center;
            border:1px solid lightgray;
            box-sizing: border-box;
            padding: 0px 20px;
        }
    </style>
</head>
<body>
    <div id="app">
        <ul class="list-group">
            <li>张三</li>
            <li>李四</li>
            <li>王五</li>
        </ul>
        <hr>
        <ul class="list-group">
            <li>张三</li>
            <li>李四</li>
            <li>王五</li>
        </ul>
        <hr>
        <ul class="list-group">
            <li>张三</li>
            <li>李四</li>
            <li>王五</li>
        </ul>
    </div>
</body>
<script src="./js/vue.js"></script>
```

```
<script>
    new Vue({
        el:"#app",
    })
</script>
</html>
```

张三

李四

王五

张三

李四

王五

张三

李四

王五

在上面的代码当中，我们可以明显的感觉到有很大的代码冗余量。这样时候我们就会想到一个封装的原理。关键的问题就在于我们的HTML是否能够封装

关于virtual DOM

在上一个章节当中，我们其实已经抛出了一个问题，这个问题主是能否对HTML标签进行封装？在Vue的内部是可能，因为Vue是支持`virtual DOM`

那到底什么是`virutal DOM`呢，具体，我们可以通过案例来说明

```
<body>
    <div id="app">
        <user-list></user-list>
        <hr>
        <user-list></user-list>
        <hr>
        <user-list></user-list>
    </div>
    <template id="temp1">
        <ul class="list-group">
            <li>张三</li>
            <li>李四</li>
            <li>王五</li>
        </ul>
    </template>
</body>
<script src="./js/vue.js"></script>
<script>
```

```
    //这就代表封装了一段html标签，这个html标签就是#temp1内部的东西
    Vue.component("user-list",{
        template:"#temp1"
    });
    new Vue({
        el: "#app",
    })
</script>
```

这个时候，我们通过上面的代码同样的实现了之前的东西

> 代码说明：
>
> 1. 我们把要封装的`html`代码放在了`<template>`这个标签里面，并定义了一个ID
> 2. 我们在JS里面使用了`Vue.component`的方法去注册了一个名叫`user-list`的虚拟标签
> 3. 在HTML的页面上面，我们使用`<user-list></user-list>`调用了一个像这样的标签
>
> 这个时候页面上面就会展示我们所封装的东西，这种使用封装的原理开发的技术我们叫`virtual DOM`

在前端的三大框架里面，都支持类似于上面的这种开发方式，这种开发方式就是基于虚拟DOM的开发方式。**而这种虚拟DOM在框架里面叫组件**,所以我们的`Vue.component`其实就是注册了一个组件

注意事项：

1. 组件的名称(`virtual DOM`名称)不能是已经存在的标签名称
2. 组件的名称(`virtual DOM`名称)不能是驼峰命名，如果非要使用驼峰命名，我们就需要使用转义

   例如我们注册的时候的组名称叫`userInfoList`，则在使用标签的时候应该是`<user-info-list>`

全局组件

> 全局组件顾名思义就是在全局范围都可以使用的

在Vue当中，vue是支持组件化开发的，在使用组件的时候，只用记住一个点就行了，所有的组件也都是vue对象

使用局组固定使用`Vue.component`的方式去注册，它的使用过程在上面已经讲解过了

1. 先准备好封装的html标签，然后放在`<template>`标签里在，并注明id

   ```
   <template id="temp1">
       <!-- 在这里准备好你要写的html标签即可-->
   </template>
   ```

   在以前的时候，还有另一种写法，这种写法大家都很熟悉

   ```
   <script type="text/template" id="temp2">
       <div>
            <h2>这是第二个组件，真哥真帅</h2>
       </div>
   </script>
   ```

   😈 **注意事项**：

   - 封装标签的时候，`template`是不能够放在托管区域内部的
   - 封装标签的时候是不支持*片段代码*，也就是`<template>`有且只允许有一个根标签

2. 在js代码里面调用`Vue.component`来注册为全局组件

```
Vue.component("组件名",{
    template:"#temp1"
})
```

> 关于组件名的要求，之前在`virutal DOM`里面已经提到过了，这里面不在赘述

3. 注册好全局组件以后，就可以在任何地方都去使用这个全局组件了

现在我们根据上面的步骤，去尝试着完成全局组件的代码案例

案例

```
<body>
    <div id="app">
        <!-- 我们在这里调用了第3个组件 -->
        <three></three>
    </div>
    <template id="temp1">
        <div>
            <h2>这是第1个部分</h2>
        </div>
    </template>
    <template id="temp2">
        <div>
            <h2>这是第2个部分</h2>
            <!-- 我们在组件内部又调用了组件 -->
            <one></one>
        </div>
    </template>
    <template id="temp3">
        <div>
            <h2>这是第3个部分</h2>
            <two></two>
        </div>
    </template>
</body>
<script src="./js/vue.js"></script>
<script>
    Vue.component("one", {
        template: "#temp1"
    });
    Vue.component("two", {
        template: "#temp2"

    });
    Vue.component("three", {
        template: "#temp3"
    });

    //上面注册了三个全局组件，全局组件应该是在任何地方都可以使用的
    new Vue({
        el: "#app"
    })
</script>
```

> 案例分析：
>
> 在上面的案例里面，我们可以看到当注册为全局组件以后，我们就可以在任何地方去调用，包括在其它组件的内部去调用这个组件

局部组件

局部组件顾名思义就是只能局部使用，它与全局组件是相对应的

局部组件本质上面就是一个对象，它定义好了这个对象以后需要在`components`这个属性下面再去注册一下才可能 使用

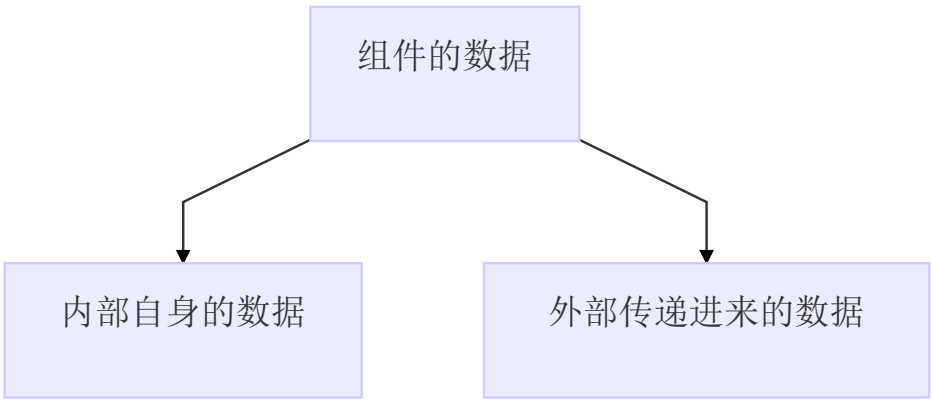
```
<body>
    <div id="app">
        <three></three>
        <one></one>
        <two></two>
    </div>
    <template id="temp1">
        <div>
            <h2>这是第1个部分</h2>
        </div>
    </template>
    <template id="temp2">
        <div>
            <h2>这是第2个部分</h2>
            <!-- 在这里，我们使用了three这个组件 -->
            <three></three>
        </div>
    </template>
    <template id="temp3">
        <div>
            <h2>这是第3个部分</h2>
        </div>
    </template>
</body>
<script src="./js/vue.js"></script>
<script>
    let one = {
        template: "#temp1"
    }

    let three = {
        template: "#temp3",

    }
    let two = {
        template: "#temp2",
        components:{
            //在这里，我们注册了three这个组件
            three
        }
    }

    new Vue({
        el: "#app",
        components: {
```

```
            one,
            two,
            three
        }
    })
</script>
```

组件中的数据

vue组件当中的数据主要来源于两个地方

1. 组件自身的数据
2. 从外部传递进来的数据



组件自身的数据

问题：怎么样在组件自身定义数据?

```
<body>
    <div id="app">
        <user-info></user-info>
        <hr>
        <user-info></user-info>
    </div>
    <template id="temp1">
        <div>
            <h2>{{stuName}}</h2>
            <h2>{{abc}}</h2>
        </div>
    </template>
</body>
<script src="./js/vue.js"></script>
<script>
    let userInfo = {
        template: "#temp1",
        data(){
            return {
                stuName:"蒋雨晴"
```

```
            }
        }
    }
    new Vue({
        el: "#app",
        data:{
            userName:"标哥哥"
        },
        components:{
            userInfo
        }
    })
</script>
```

在上面的案例当中，我们看到组件自身是可以定义数据的，但是它定义数据的时候有一点点与之前不一样，这的`data`是一个方法，这个方法再返回出一个对象

```
data(){
    return {

    }
}
```

组件从外部接收数据

> 本质上来说就是父级组件向子级件件传递数据

vue父子组件之前的传值是通过**自定义属性**来完成的

```
<body>
    <div id="app">
        <user-info :age="22" :teacher-name="teacherName"></user-info>
    </div>
    <template id="temp1">
        <div>
            <h2>AAAAA</h2>
            <h1>{{age}}</h1>
            <h1>{{teacherName}}</h1>
        </div>
    </template>
</body>
<script src="./js/vue.js"></script>
<script>
    let userInfo = {
        template: "#temp1",
        // 接收外部传递进来的数据
        props:{
            age:{
                type:Number,
                default:35
            },
            teacherName:{
                type:String,
                //必须要从父级接收一个像这样的值，否则报错
                required:true
            }
```

```
        }
    }
    new Vue({
        el: "#app",
        data:{
           teacherName:"标哥哥"
        },
        components:{
            userInfo
        }
    })
</script>
```

> 代码说明：
>
> 1. 在上面有个小小的注意事项，`age="22"`传过去的是一个字符串的22，如果我们写成`:age="22"`则传递过去的是number类型的22
>
> 依次类推，如果我们想传一个布尔类型的`true`应该写成`:isShow="true"`
>
> 2. 在进行自定义属性传值的时候，我们要尽量不要使用驼峰命名，如果非要使用，则请进行转义
> 3. 在定义的接收值的过程当中，可能通过`type`去限制它的数据类型，可以通过`default`去设置它默认的值
> 4. 如果某一个值必须要由父级传递下来，则可以使用`required:true`来设置，默认情况下是`false`

上面的接值是一个完整的写法，它还有一个简写的方式，如下

```
let userInfo = {
    template: "#temp1",
    // 接收外部传递进来的数据
    props:["age","teacherName"]
}
```

组件中的事件与方法

组件我们可以看成是一个小型的`vue`，它有自己的托管区域，它有自己的数据，同理它也有自己的方法（所有组件内部是有自己的事件方法的）

```
<body>
    <div id="app">
        <temp1></temp1>
    </div>
    <template id="temp1">
        <div>
            <h2>{{abc}}</h2>
            <button type="button" @click="changeAbc">改变abc的值</button>
        </div>
    </template>
</body>
<script src="./js/vue.js"></script>
<script>
    let temp1 = {
        template:"#temp1",
```

```
        data(){
            return {
                abc:"hello world"
            }
        },
        methods:{
            changeAbc(){
                this.abc = "无敌帅气的标哥哥";
            }
        }
    }
    new Vue({
        el: "#app",
        components:{
            temp1
        }
    })
</script>
```

组件内部的方法归组件内部使用

父级组件调用子级组件的方法

在vue以及vue组件的内部，我们可以通过两个属性向上或向下去寻找

1. `$children`当前组件下面的使用了的所有组件
2. `$parent`当前组件的父级组件


```
<body>
    <div id="app">
        <button type="button" style="color: red;" @click="myChange">我在外边能不能调用temp1组件changeAbc方法呢</button>
        <temp1></temp1>
    </div>
    <template id="temp1">
        <div>
            <h2>{{abc}}</h2>
            <button type="button" @click="changeAbc">改变abc的值</button>
        </div>
    </template>
</body>
<script src="./js/vue.js"></script>
<script>
    let temp1 = {
        template:"#temp1",
        data(){
            return {
                abc:"hello world"
            }
```

```
        },
        methods:{
            changeAbc(){

                this.abc = "无敌帅气的标哥哥";
            }
        }
    }
    new Vue({
        el: "#app",
        methods:{
            myChange(){
                //这个时候我们就可以通过这样一种方式来调用
                this.$children[0].changeAbc();
            }
        },
        components:{
            temp1
        }
    })
</script>
```

在上面的代码当中是不合格也是不严谨的，因为我们通过`this.$children`再通过索引去找的时候是会存在一定的隐患的，当索引不对，一切白费

这个时候我们就可以结合我们之前所学习的`ref`来完成

```
<body>
    <div id="app">
        <button type="button" style="color: red;" @click="myChange">我在外边能不能调用temp1组件changeAbc方法呢</button>
        <!-- 关键就在这里使用了ref -->
        <temp1 ref="temp1"></temp1>
    </div>
    <template id="temp1">
        <div>
            <h2>{{abc}}</h2>
            <button type="button" @click="changeAbc">改变abc的值</button>
        </div>
    </template>
</body>
<script src="./js/vue.js"></script>
<script>
    let temp1 = {
        template:"#temp1",
        data(){
            return {
                abc:"hello world"
            }
        },
        methods:{
            changeAbc(){

                this.abc = "无敌帅气的标哥哥";
            }
        }
    }
```

```
    new Vue({
        el: "#app",
        methods:{
            myChange(){
                //在这里通过$refs找到之前的virtual DOM，然后再调用这个组件里面方法
                this.$refs.temp1.changeAbc();
            }
        },
        components:{
            temp1
        }
    })
</script>
```

数据流的单向性

vue当中的数据流的方向是自顶向下产生的，不可逆，这种现象我们称之为数据流的单向性，具体的表现为父级组件的数据传递给子级组件以后子级组件是不可以擅自更改的，但是父级组件改变数据子级组件接收的这个数据会自动更改

```
<body>
    <div id="app">
        <h1>我是杨标，我女儿的名子叫------{{daughterName}}</h1>
        <button type="button" @click="fatherChangeName">我是父亲，我来改女儿的名子
</button>
        <abc :daughter-name="daughterName"></abc>
    </div>
    <template id="temp1">
        <div>
            <h2>我是标哥的女儿，我叫*************{{daughterName}}</h2>
            <hr>
            <button type="button" @click="changeName">我要改名子</button>
        </div>
    </template>
</body>
<script src="./js/vue.js"></script>
<script>
    let abc = {
        template:"#temp1",
        props:["daughterName"],
        methods:{
            changeName(){
                this.daughterName = "杨柳彤";
            }
        }
    }
    new Vue({
        el: "#app",
        data: {
            daughterName: "杨妞"
        },
        methods:{
            fatherChangeName(){
                this.daughterName = "杨柳彤";
            }
        },
        components:{
```

```
            abc
        }
    })
</script>
```

> **代码分析**:
>
> 1. 在上面的案例当中，我们发现如果子级组件想改变父级组件传递过来的值的时候是不可能的，会直接报错
> 2. 当我们在父级更改了这个`daughterName`以后，子级组件也跟着改变了，这就是数据流的单身性的体现

数据流的单向性是保证了数据的安全

破坏数据流的单向性

vue里面如果要破坏数据流的单向性这个特点，我们可以从2个方面去发现

1. 使用`const`锁栈不锁堆的原理来完成
2. 使用自定义事件去完成

使用`const`的原理去完成

先看如下代码

```
/*
const daughterName = "杨妞";
daughterName = "杨柳彤";
*/

//上面的现的像是属于const常量的现象，const锁栈， 不锁堆
//vue的单向数据流也是这样的，所以......
const obj = {
    daughterName: "杨妞"
}

obj.daughterName = "杨柳彤";
console.log(obj.daughterName);
```

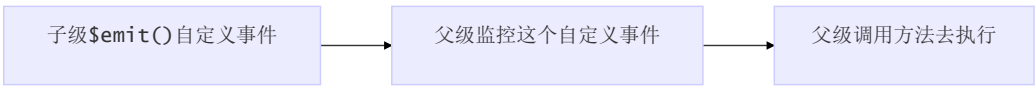
在上面的代码当中很好的解释了为什么直接对`daughterName`赋值，而对`obj`属性赋值又可以，因为它锁栈不锁堆，而改变的又是堆里面的数据

现在将这个特点应用于`vue`当中去

```
<body>
    <div id="app">
        <h1>我是杨标，我女儿的名子叫------{{obj.daughterName}}</h1>
        <button type="button" @click="fatherChangeName">我是父亲，我来改女儿的名子
</button>
        <abc :obj="obj"></abc>
    </div>
    <template id="temp1">
        <div>
            <hr>
            <h2>我是标哥的女儿，我叫**********{{obj.daughterName}}</h2>
            <button type="button" @click="changeName">我要改名子</button>
        </div>
```

```
    </template>
</body>
<script src="./js/vue.js"></script>
<script>
    let abc = {
        template:"#temp1",
        props:["obj"],
        methods:{
            changeName(){
                this.obj.daughterName = "杨柳彤";
            }
        }
    }
    new Vue({
        el: "#app",
        data: {
            obj:{
                daughterName: "杨妞"
            }
        },
        methods:{
            fatherChangeName(){
                this.obj.daughterName = "杨柳彤";
            }
        },
        components:{
            abc
        }
    })
</script>
```

使用自定义事件去完成




```
<body>
    <div id="app">
        <h1>我是杨标，我女儿的名子叫------{{daughterName}}</h1>
        <abc :daughter-name="daughterName" @dadchangemyname="dadChangeMyName">
</abc>
    </div>
    <template id="temp1">
        <div>
            <h2>我是标哥的女儿，我叫**********{{daughterName}}</h2>
            <hr>
            <button type="button" @click="changeName">我要改名子</button>
        </div>
    </template>
</body>
```

```
<script src="./js/vue.js"></script>
<script>
    let abc = {
        template: "#temp1",
        props: ["daughterName"],
        methods: {
            changeName() {
                // 如果直接去改，这是不可行的
                //this.daughterName = "杨柳彤";
                //正常的思维逻辑应该是通过父级组件去改更某个值
                //this.$emit相当于主动的去触发一个事件
                this.$emit("dadchangemyname", "杨柳彤");
            }
        }
    }
    new Vue({
        el: "#app",
        data: {
            daughterName: "杨妞"
        },
        methods: {
            dadChangeMyName(newName) {
                if (newName.startsWith("杨")) {
                    this.daughterName = newName;
                }
                else{
                    alert("不行，你必须姓杨");
                }
            }
        },
        components: {
            abc
        }
    })
</script>
```

组件中的插槽

普通插槽

组件中的插槽可以理解成我们之前的电脑里同的主板，这个主板可以确定大多数的功能，对于不确定的东西则预留一个插槽放在那里，方便后期插入东西进去

在我们之前学习封装组件开始，我们就已经有知道组件是可以把HTML代码封闭在一起的，但是后期在封装的过程当中经常会有组件当中确定的内容，这个时候我们就要借助于今天所学习的插槽去完成

```
<body>
    <div id="app">
        <user-info>
            <input type="text" placeholder="请输入内容啊">
        </user-info>
    </div>
    <template id="temp1">
        <div class="box">
            <hr>
            <!-- 在这里有个内容确定不了怎么办呢？
                这里就要预留一个接口，方便后期插入数据
```

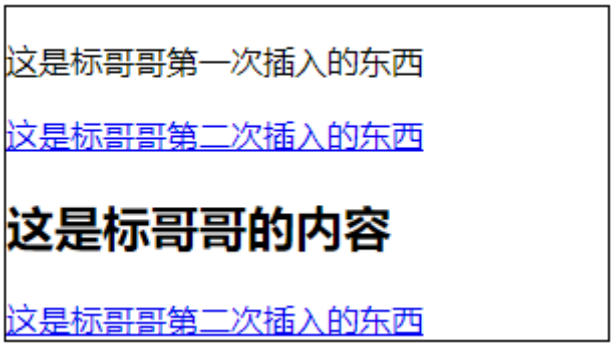
```
            -->
            <slot></slot>
            <h2>这是标哥哥的内容</h2>
            <slot></slot>
        </div>
    </template>
</body>
<script src="./js/vue.js"></script>
<script>
    Vue.component("user-info", {
        template: "#temp1"
    });
    new Vue({
        el: "#app"
    })
</script>
```

> 代码分析：
>
> 1. 插槽是在调用这个组件的时候在中间部分插入进入的
> 2. 插槽在组件封装的时候使用`<slot>`去标记，在组件内部可能出现多个的`<slot>`标记

具名插槽

我们上面的普通插槽里面都看到了，当我们在向组件内部插入东西的时候，所以具有`<slot>`的地方都被插入了内容。这种情况下其实还可以设置指定的名称以方便插入到指定的位置，这种现象叫具名插槽

```
<body>
    <div id="app">
        <user-info>
            <p slot="header">这是标哥哥第一次插入的东西</p>
            <a slot="footer" href="#">这是标哥哥第二次插入的东西</a>
        </user-info>
    </div>
    <template id="temp1">
        <div class="box">
            <slot name="header"></slot>
            <slot name="footer"></slot>
            <h2>这是标哥哥的内容</h2>
            <slot name="footer"></slot>
        </div>
    </template>
</body>
<script src="./js/vue.js"></script>
<script>
    Vue.component("user-info", {
        template: "#temp1"
    });
    new Vue({
        el: "#app"
    })
</script>
```



> 代码说明：
>
> 1. 我们在组件里面定义插槽的时候使用了`<slot>`标签，并且在这个标签上面定义了`name`这个属性，这就是具名插槽
> 2. 在调用这个组件的时候，可以向指定的插槽位置插入内容，只需要在这个元素上面添加`slot="名称"`即可
> 3. 具名插槽也是可以多次使用的，所以在上面的代码当中`name="footer"`这个插槽就出现2次

案例：下面就是一个典型的组件封装中使用了插槽的案例

```
<!DOCTYPE html>
<html lang="zh">

<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>组件插槽案例</title>
    <link rel="stylesheet"
href="https://at.alicdn.com/t/font_2298735_gwfdo52sccf.css">
    <style>
        * {
            margin: 0;
            padding: 0;
            list-style-type: none;
        }

        .page-header {
            height: 45px;
            background-color: #008DE1;
            display: flex;
            flex-direction: row;
            justify-content: center;
            align-items: center;
            color: white;
            position: relative;
        }

        .page-header .left-back {
            position: absolute;
            left: 10px;
        }

        .page-header .right-menu {
            position: absolute;
            right: 10px;
        }
```

```
    </style>
</head>

<body>
    <div id="app">
        <page-header :show-back="true">
            <span class="iconfont iconyonghu_huaban1"></span>
        </page-header>
        <hr>
        <page-header title="菜品详细【香干回锅肉】">
            <span class="iconfont iconicon-test"></span>
        </page-header>
        <hr>
        <page-header title="登陆">
            <span class="iconfont iconcaidan"></span>
        </page-header>
        <hr>
        <page-header :show-back="true" title="标哥哥">
            <span class="iconfont iconheart"></span>
            <span class="iconfont iconshouquan"></span>
        </page-header>
    </div>
    <template id="temp1">
        <div class="page-header">
            <div class="left-back" v-if="showBack">
                <span class="iconfont iconfanhui1"></span>
                返回
            </div>
            <span>{{title}}</span>
            <div class="right-menu">
                <slot></slot>
            </div>
        </div>
    </template>
</body>
<script src="./js/vue.js"></script>
<script>
    Vue.component("page-header", {
        template: "#temp1",
        props: {
            showBack: {
                type: Boolean,
                default: false
            },
            title:{
                type:Object,
                default:"软帝点餐"
            }
        },
        data() {
            return {

            }
        }
    })
    new Vue({
        el: "#app"
    })
```

```
</script>

</html>
```

作用域插槽

作用域插槽是一种比较我常见的现象，这种现象主要体现在插槽里面的数据来源的问题，我们还是通过代码的角度 去理解

```
<body>
    <div id="app">
        <user-info>
            <h1 slot="aaa">{{userName}}</h1>
            <h1 slot="bbb" slot-scope="scope">
                地址：{{scope.addr}}
                年龄：{{scope.age}}
            </h1>
        </user-info>
    </div>
    <template id="temp1">
        <div>
            <slot name="aaa"></slot>
            <h2>哈哈哈，我是一个组件啦</h2>
            <slot name="bbb" :addr="addr" :age="age"></slot>
        </div>
    </template>
</body>
<script src="./js/vue.js"></script>
<script>
    Vue.component("user-info",{
        template:"#temp1",
        data(){
            return {
                addr:"湖北省武汉市",
                age:18
            }
        }
    });
    new Vue({
        el: "#app",
        data:{
            userName:"张三"
        }
```

```
    })
</script>
```

> 代码分析：
>
> 1. 在`slot="aaa"`的这个插槽里面，里面的数据是来源于外边的`#app`的，这一点说明插槽里面的数据默认是受外部控制的【所以插槽里面的数据，事件方法等都应该来源于外部`#app`】
> 2. 在`slot="bbb"`的这个插槽里面，我们添加了一个`slot-scope`，这样就从组件的内部获取了作用域，然后就可以从组件内部拿到传递出来的数据
> 3. 在封装组组件`user-info`的时候，我们在html代码里面，向外边传递了两个数据分别是`addr`与`age`，最终这两个变量都会在同一个作用域里面传递出去【它会形成一个使用域对象传递scope出来】

上面的`slot-scope`是以前旧的写法，现在有新写法叫`v-slot`了

```
<template id="temp1">
        <div>
            <slot name="aaa"></slot>
            <h2>哈哈哈，我是一个组件啦</h2>
            <slot name="bbb" :addr="addr" :age="age"></slot>
        </div>
</template>
```

旧版本的语法

```
<user-info>
    <h1 slot="aaa">{{userName}}</h1>
    <h1 slot="bbb" slot-scope="scope">
        地址：{{scope.addr}}
        年龄：{{scope.age}}
    </h1>
</user-info>
```

新版本的语法

```
<user-info>
    <h1 slot="aaa">{{userName}}</h1>
    <template v-slot:bbb="scope">
        地址：{{scope.addr}}
        年龄：{{scope.age}}
    </template>
</user-info>
```

Vue及组件的生命周期

组件可以看成是一个小型的vue文件，它内部的原理与vue的原理是一样的，所以我们如果理解了组件的生命周期以后对于vue的生命周期来说是一样的

组件从我们最初创建开始，它就会有一个生成过程，这个过程我们怎么看呢？

通俗一点来说，我们经常会把vue分为4个过程，8个状态，它不在不同的状态和不同的过程当中都会执行相同的操作，这个相应的操作我们叫**钩子函数**

钩子函数：所谓的钩子函数指的是vue在不同的生命周期下面自己调用的函数

1. `beforeCreate` vue创建以前
2. `created` vue创建以后
3. `beforeMount` vue挂载之前
4. `mounted` vue挂载之后
5. `beforeUpdate` vue内部更新以前
6. `updated` vue内部更新以后
7. `beforeDestory()` vue销毁之前
8. `destoryed()` vue销毁之后

在vue的4个过程8个状态当中一定要注意，不同的阶段是可以执行不同的操作

周期名称	data属性	computed属性	methods	$refs
`beforeCreate()`	否	否	否	否
`created()`	是	是	是	否
`beforeMounte()`	是	是	是	否
`mounted()`	是	是	是	是

跨生命周期的调用

vue在不同的生命周里面执行不同的操作这样做确实让我们的代码会更清晰，但同也给我们带来了一些不便

```
<body>
    <div id="app">
        <h2 ref="pageTitle">{{userName}}</h2>
        <button type="button" @click="userName = '李四四'">更改数据</button>
    </div>

</body>
<script src="./js/vue.js"></script>
<script>
    new Vue({
```

```
        el: "#app",
        data: {
            userName: "张珊"
        },
        created() {
            let num = parseInt(Math.random() * 1000);
            console.log(num);
            //我就想在这里操作dom怎么办呢
            this.$nextTick(() => {
                this.$refs.pageTitle.innerText = num;
            });
        },

    })
</script>
```

在上面的代码当中，我们就发现了一个点，本来在`created`里面， 我们是不能调使用`$refs`的操作的，但是我们通过`$nextTick`来实现了这个跨生命周期的操作

`$nextTick`的作用就是让回调函数里面的代码在合适的生命周期里面自动调用

v-if与v-show的区别

之前提到过的vue的四个过程八个状态里面，我们有创建，也有销毁，现在我们来体验一下它的创建与销毁过程

v-show的隐藏

```
<body>
    <div id="app">
        <button type="button" @click="flag=!flag">切换显示与隐藏</button>
        <user-info v-show="flag"></user-info>
    </div>
    <template id="temp1">
        <div>
            <input type="text" v-model="userName">
            <h2>{{userName}}</h2>
        </div>
    </template>
</body>
<script src="./js/vue.js"></script>
<script>
    Vue.component("user-info", {
        template: "#temp1",
        data() {
            return {
                userName: "张珊"
            }
        },
        created() {
            console.log("我是创建的生命周期钩子函数created");
        }
    })
    new Vue({
        el: "#app",
        data: {
            flag: true
```

```
        }

    })
</script>
```

通过上面的案例，我们发现，组件内部的`created`钩子函数只执行了一次，所以`v-show`只是让这个元素不可见，并没有销毁

v-if的情况

```
<body>
    <div id="app">
        <button type="button" @click="flag=!flag">切换显示与隐藏</button>
        <user-info v-if="flag"></user-info>
    </div>
    <template id="temp1">
        <div>
            <input type="text" v-model="userName">
            <h2>{{userName}}</h2>
        </div>
    </template>
</body>
<script src="./js/vue.js"></script>
<script>
    Vue.component("user-info", {
        template: "#temp1",
        data() {
            return {
                userName: "张珊"
            }
        },
        created() {
            console.log("我是创建的生命周期钩子函数created");
        }
    })
    new Vue({
        el: "#app",
        data: {
            flag: true
        }
    })
</script>
```

这个时候我们看到组件内部的`created`生命周期的钩子函数在不停的去执行，所以`v-if`的不可见是把组件销毁掉了，它会不停的去执行生命周期的

`keep-alive`的使用

vue与组件从创建到销毁是有一个过程的，所以是不是所有的组件都会销毁呢?

答案是否定的!

在vue的内部，有一个自带的组件叫`<keep-alive>`在这个组件下面的内容是不会被销毁掉的，也就是不会执行`destory`这个过程

但是它多了另外的两个钩子函数

1. `activated`当激活【显示】的时候触发

2. `deactivated`当休眠【隐藏】的时候触发

```
<body>
    <div id="app">
        <button type="button" @click="flag=!flag">切换显示与隐藏</button>
        <keep-alive>
            <user-info v-if="flag"></user-info>
        </keep-alive>
    </div>
    <template id="temp1">
        <div>
            <input type="text" v-model="userName">
            <h2>{{userName}}</h2>
        </div>
    </template>
</body>
<script src="./js/vue.js"></script>
<script>
    Vue.component("user-info", {
        template: "#temp1",
        data() {
            return {
                userName: "张珊"
            }
        },
        //创建的生命周期
        created() {
            console.log("我是创建的生命周期钩子函数created"+Math.random());
        },
        activated(){
            console.log("activated----------");
        },
        deactivated(){
            console.log("deactivated----------");
        }
    })
    new Vue({
        el: "#app",
        data: {
            flag: true
        }
    })
</script>
```

我们在之前的组件上面加了一个`<keep-alive>`，这样，这个组件即使在使用`v-if`隐藏的时候也不会销毁了，它只会有另外的两个生命周期中来回