# 第六节 组件化开发

## 1. 生命周期

### 1.1 生命周期概述

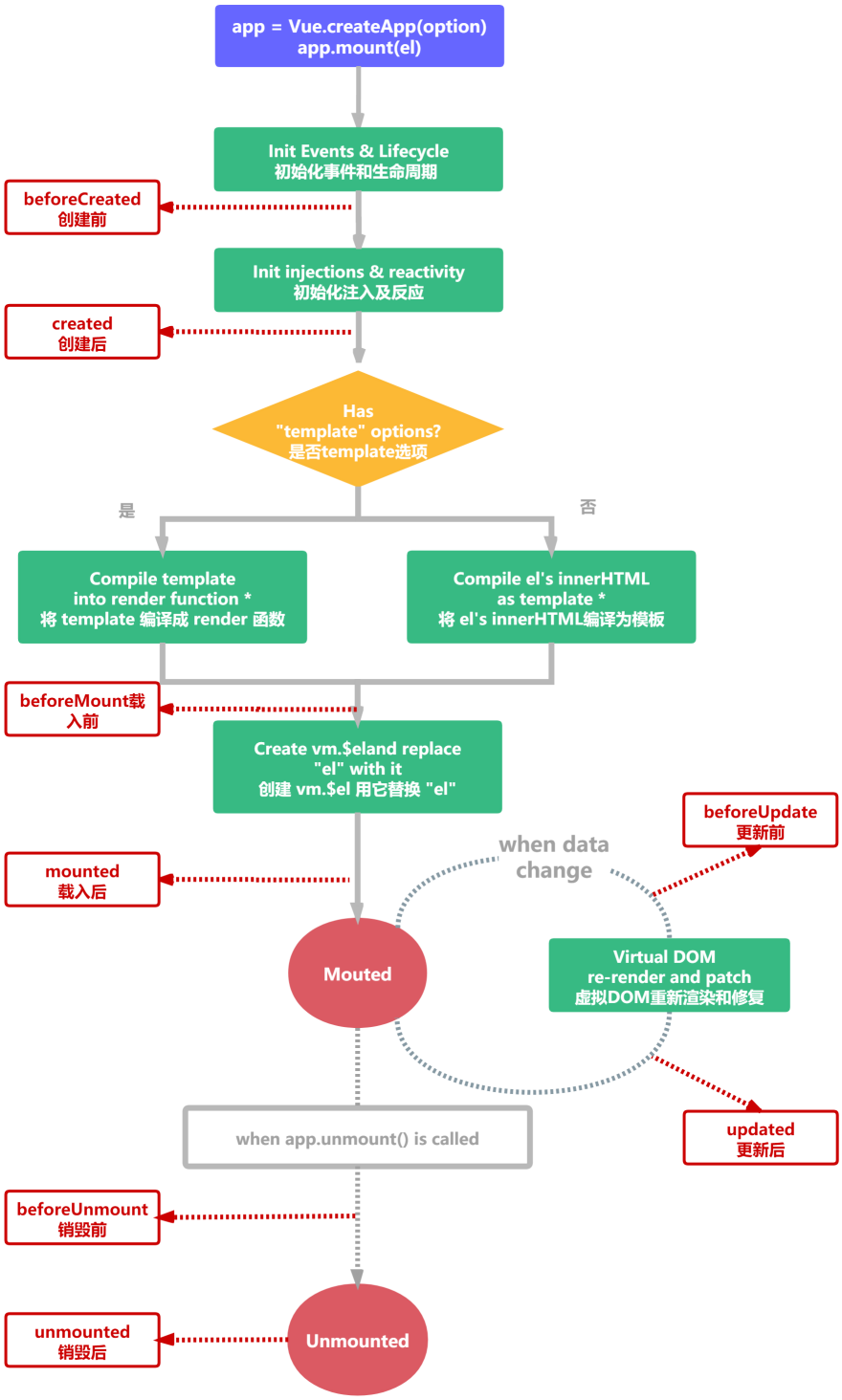
Vue中，**每个组件**在被创建时都要经过一系列的初始化过程。例如，设置数据监听、编译模板、将实例挂载到DOM、在数据变化时更新DOM等。我们把Vue 3.0的App从创建到销毁的过程称为生命周期，在这个过程中运行的一些函数称为生命周期的钩子函数。这些钩子函数为我们在Vue的生命周期不同阶段进行程序控制提供了可能。

Vue3.0的生命周期主要分成**四个阶段**，分别是`create`（初始创建）、`mount`（挂载）、`update`（更新）和`destroy`（销毁）。每个阶段均提供了强大的钩子函数，如下表所示：

| 序号 | 选项式 API（Vue2） | 组合式API（Vue3） | 说明 |
| --- | --- | --- | --- |
| 1 | beforeCreate | Not needed* | 用setup取代 |
| 2 | created | Not needed* | 用setup取代 |
| 3 | beforeMount | onBeforeMount | 在实例挂载之前被调用 |
| 4 | mounted | onMounted | 在实例挂载完成之后被调用 |
| 5 | beforeUpdate | onBeforeUpdate | 在数据发生改变后，DOM 被更新之前被调用 |
| 6 | updated | onUpdated | 在数据发生改变后且DOM 被更新之后被调用 |
| 7 | beforeUnmount | onBeforeUnmount | 在卸载组件实例之前调用 |
| 8 | unmounted | onUnmounted | 在卸载组件实例之后调用 |

| 序号 | 选项式 API（Vue2） | 组合式API（Vue3） | 说明 |
| --- | --- | --- | --- |
| 9 | errorCaptured | onErrorCaptured | 在捕获一个来自后代组件的错误时被调用 |
| 10 | renderTracked | onRenderTracked | 当虚拟 DOM 重新渲染被跟踪时调用 |
| 11 | renderTriggered | onRenderTriggered | 当虚拟 DOM 重新渲染被触发时调用 |
| 12 | activated | onActivated | 被 keep-alive 缓存的组件激活时调用 |
| 13 | deactivated | onDeactivated | 被 keep-alive 缓存的组件失活时调用 |

其生命周期及钩子函数的运行时机说明如下图所示（该图是`vue2`的，官网没有最新的）：




上图所表示的含义如下：

（1）`Vue.createApp(options)`：根据options（选项）的要求创建Vue的应用。

（2）`init events&lifecycle`：执行一些与初始化和生命周期相关的操作。

（3）`init injections&reactivity`：初始化注入和校验。

（4）`setup`：完成组件实例创建并且属性已经绑定，但是DOM还没有生成。

（5）`Has template option?`：判断是否存在template选项，如果有template选项，将使用render()函数进行治染；如果没有模板，将外部的HTML作为模板进行编译。

（6）`onBeforeMount`：App挂截之前该函数被调用，onBeforeMount之前el还是undefined。

（7）`Create app.$el and replace el with it`：给Vue应用对象添加el成员，并且替换挂载的DOM元素。

（8）`onMounted`：组件挂载到页面之后执行的句子函数，该函数可以用来向后端发起请求并取回数据等。

（9）`onBeforeupdate`：是可以监听到数据变化的钩子函数，但是该函数是在数据变化之前被触发，也就是视图层并没有被重新渲染，视图层的数据也并没有变化。

（10）`onUpdated`：是可以监听到数据变化的钩子函数，该函数是在数据变化之后被触发，也就是视图层被重新渲染，视图层的数据被更新。

（11）`onBeforeUnmount`：该函数在App组件被销毁之前调用。

（12）`onUnMountd`：该函数在Vue的App被销毁后调用。调用后，Vue的App指示的所有东西都会解绑定，所有的事件监听器会被移除，所有的子组件也会被销毁。

## 1.2 钩子函数的使用

### 1. 钩子函数的使用

生命周期函数也称为钩子函数，钩子函数只能在setup()方法中使用。使用钩子函数前，必须从Vue中引入。例如，引入onMounted钩子函数的语句如下：

```
1 import { onMounted } from 'vue'
```

然后，在setup()中定义该钩子函数（onMounted）运行时完成的任务。其代码如下所示：

```
setup(){
    onMounted(()=>{
        //组件加载后，需要完成的任务
    })
}
```

**2. 钩子函数的执行顺序**

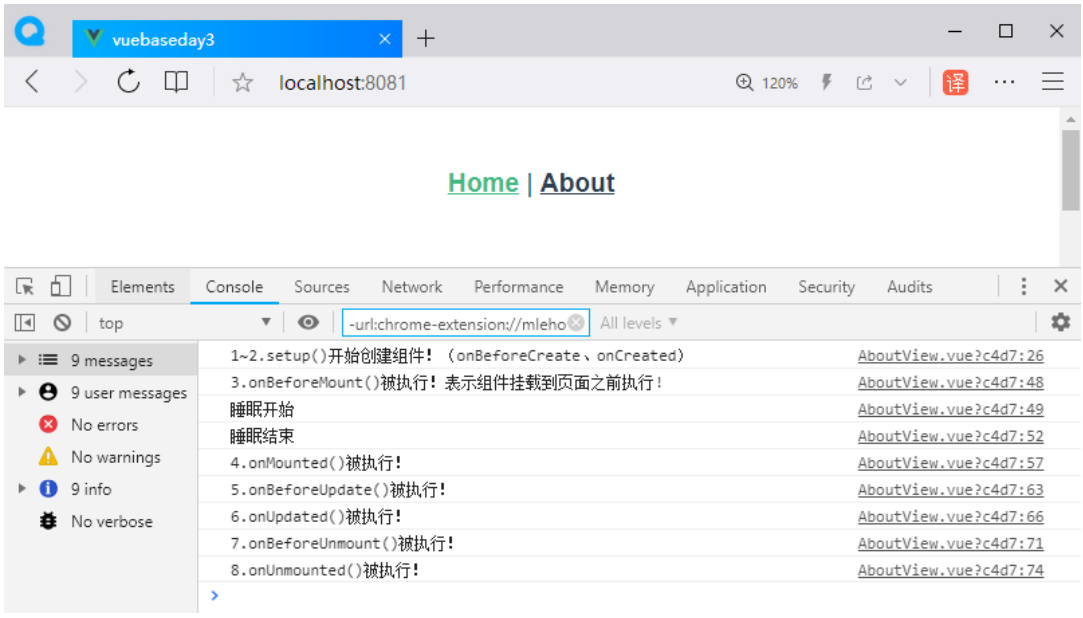
在脚手架项目中，编辑根组件About.vue，可以观察钩子函数执行的顺序，代码如下：

```
<template>
  <div class="about">
    <h1>This is an about page</h1>
    <div>
      <input type="radio" value="1" v-model="sex" />男
      <input type="radio" value="2" v-model="sex" />女
    </div>
  </div>
</template>

<script>
import {
  reactive,
  toRefs,
  onBeforeMount,
  onMounted,
  onBeforeUpdate,
  onUpdated,
  onBeforeUnmount,
  onUnmounted,
  onDeactivated
} from 'vue'
export default {
  setup() {
```

```
25      console.log('1~2.setup()开始创建组件！ (onBeforeCreate、onCreated) ')
26      const state = reactive({
27        sex: 2,
28        count: 0
29      })
30
31      // 1.在实例生成之前会自动执行的函数
32      // onBeforeCreate(() => console.log('1.onBeforeCreate()被执行！ '))
33
34      // 2.在实例生成之后会自动执行的函数
35      // onCreated(() => console.log('2.onCreated()被执行！ '))
36
37      // 3.1 利用一个伪死循环阻塞主线程。因为JS是单线程的.
38      function sleep(delay) {
39        var start = new Date().getTime()
40        while (new Date().getTime() - start < delay) {
41          continue
42        }
43      }
44
45      // 3.组件内容被渲染到页面之前，自动执行的函数
46      onBeforeMount(() => {
47        console.log('3.onBeforeMount()被执行！表示组件挂载到页面之前执行!')
48        console.log('睡眠开始')
49        // sleep(10000)
50        sleep(10)
51        console.log('睡眠结束')
52      })
53
54      // 4.组件内容被渲染到页面之后，自动执行的函数
55      onMounted(() => {
56        console.log('4.onMounted()被执行！ ')
57      })
58
59      // ▲▲▲ 改变性别选项，可以观察5、6
```

```
60
61      // 5.组件被更新前，自动执行的函数
62      onBeforeUpdate(() => console.log('5.onBeforeUpdate()被执
    行! '))
63
64      // 6.组件被更新后，自动执行的函数
65      onUpdated(() => console.log('6.onUpdated()被执行! '))
66
67      // ▲▲▲ 离开组件，可以观察7、8
68
69      // 7.组件取消挂载前，自动执行的函数
70      onBeforeUnmount(() => console.log('7.onBeforeUnmount()被执
    行! '))
71
72      // 8.组件取消挂载后，自动执行的函数
73      onUnmounted(() => console.log('8.onUnmounted()被执行! '))
74
75      const router = useRouter()
76      return {
77        ...toRefs(state)
78      }
79    }
80  }
81  </script>
```

运行程序后，在浏览器中显示结果如图7-1所示。



当依次点击About-->男--->Home时，出现的结果如上图所示。

## 2. 组件的使用步骤

所谓的`组件化`，就是把页面拆分成多个组件，每个组件单独使用`HTML`、`CSS`、`JavaScript`、`模板`、`图片`等资源进行开发与维护，然后在网页制作过程中根据需要调用相关的组件。因为组件是资源独立的，所以组件在系统内部可复用，组件和组件之间可以嵌套。如果项目比较复杂，可以极大的简少代码量，并且对后期需求的变化和维护也更加友好。

在开发一个Web项目过程中，每个网页可能会有`页头`、`侧边栏`、`导航`等区域，把多个网页中这些统一的内容定义成一个组件，可以在使用的地方像搭积木一样快速创建网页。

组件的使用分为**三个步骤**：

- `定义组件`
- `导入并注册子组件`
- `使用子组件`

### 2.1 定义组件

先定义两个组件：`ChildComp`、`FatherComp`，实现的功能为简单的按钮计数器。

（1）在脚手架的`components`文件夹下，创建子组件`ChildComp.vue`，该子组件的内容如下：

```
<template>
  <div>
    <button @click="count++">{{ count }}</button>这里是子组件
  </div>
</template>

```

```
<script>
import { reactive, toRefs } from 'vue'

export default {
  setup() {
    const state = reactive({
      count: 0
    })

    return {
      ...toRefs(state)
    }
  }
}
</script>
```

即：在`ChildComp`组件中，定义了一个按钮，点击该按钮上`数字+1`。

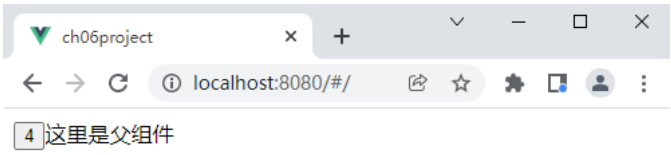
（2）在脚手架的`components`文件夹下，创建子组件`FatherComp.vue`，该子组件的内容如下：

```
<template>
  <div>
    <button @click="count++">{{ count }}</button>这里是父组件
  </div>
</template>

<script>
import { reactive, toRefs } from 'vue'

export default {
  setup() {
    const state = reactive({
      count: 0
    })

    return {
      ...toRefs(state)
```

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

即：在 FatherComp 组件中，也定义了一个按钮，点击该按钮上 数字+1 。

（3）把 FatherComp 组件作为根组件在 main.js 入门文件中进行挂载，显示结果入下图所示：




目前， FatherComp 组件和 ChildComp 组件之间，没有任何关系。

## 2.2 导入与注册子组件

在 FatherComp 组件中，导入 ChildComp 组件，并使用 components 注册该组件，修改后关键代码如下：

```
<script>
import { reactive, toRefs } from 'vue'
import ChildComp from './ChildComp.vue'      //2.1.导入子组件
export default {
  components: {
    ChildComp                                //2.2.注册子组件
  },
  setup() {
    const state = reactive({
      count: 0
    })

    return {
      ...toRefs(state)
    }
  }
```

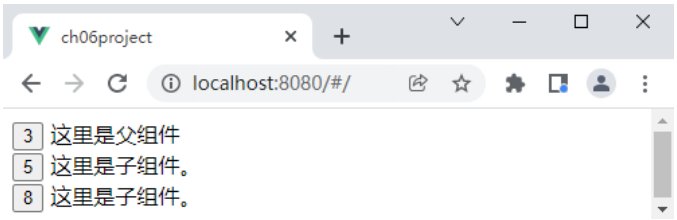
```
}
</script>
```

## 2.3 使用子组件

在`FatherComp`组件中，使用子组件，修改后关键代码如下：

```
<template>
  <div>
    <button @click="count++">{{ count }}</button> 这里是父组件
    <ChildComp></ChildComp><!--3 使用子组件-->
    <ChildComp></ChildComp>
  </div>
</template>
```

运行后，显示结果如下图所示：




在`FatherComp`组件中，重复两次使用了子组件，然后点击不同的按钮时，每个组件各自独立维护count。因为每使用一次组件，就会有一个新的实例被创建。

# 3. 组件之间的数据传递

Vue的组件传值分为三种方式：

- `父组件传递数据给子组件`：子组件通过`props`属性，获取父组件传递的数据
- `子组件传递数据给父组件`：子组件通过事件给父组件发送消息
- `跨组件传值`：这种方式我们在Vue项目实战中介绍

## 3.1 父组件传递数据给子组件

当子组件在父组件中当标签使用时，给这个标签（也就是子组件）定义一些自定义属性，值为想要传递给子组件的数据。

在子组件通过`props`属性接收父组件传递过来的数据，特别特别强调的是，`props`是专门用来接收外部数据的，该属性有`两种`接收数据的方式，分别是`数组`和`对象`，其中对象可以限制数据的类型。

父组件向子组件传递数据时，子组件`不允许`更改父组件的数据（也就是说子组件拿到的数据是只读的），因为父组件可能会同时向多个子组件传值，如果某个子组件对父组件的数据进行了修改，很可能会导致其他组件发生错误，很难对数据的错误进行追踪。

例如：通过for循环把以下对象列表数据，传递给子组件进行列表展示。

```
[
    {id: 1,realName: '张三'},
    {id: 2,realName: '李四'},
    {id: 3,realName: '王五'},
    {id: 4,realName: '麻六'}
]
```

把前面开发的两个组件`ChildComp`、`FatherComp`备份一下，我们再在上述的基础上修改。

（1）修改`ChildComp`子组件。修改后，完整代码入如下所示：

```
<template>
  <div>编号: {{ id }} , 真实姓名: {{ realName }}</div>
</template>

<script>
export default {
  props: ['id', 'realName']
}
</script>
```

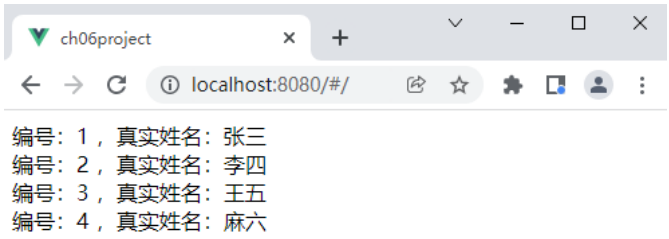
说明：`props`接收到的变量，可以在页面上直接使用。

（2）修改`FatherComp`子组件。修改后，完整代码入如下所示：

```
<template>
  <div>
    <!--3 使用子组件-->
    <ChildComp
      v-for="(obj, index) in listObj"
      :key="index"
      :id="obj.id"
      :realName="obj.realName"
    ></ChildComp>
  </div>
</template>

<script>
import { reactive, toRefs } from 'vue'
import ChildComp from './ChildComp.vue' //2.1.导入子组件
export default {
  components: {
    ChildComp //2.2.注册子组件
  },
  setup() {
    const state = reactive({
      count: 0,
      listObj: [
        { id: 1, realName: '张三' },
        { id: 2, realName: '李四' },
        { id: 3, realName: '王五' },
        { id: 4, realName: '麻六' }
      ]
    })

    return {
      ...toRefs(state)
    }
  }
}
```

```
</script>
```

（3）运行后，界面如下图所示：

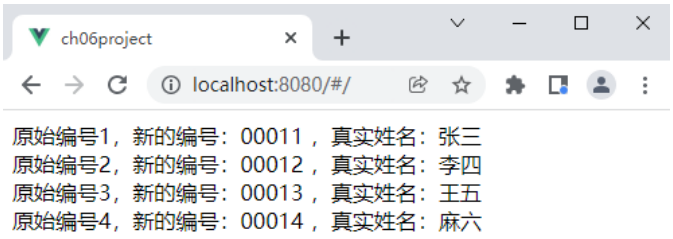



（4）`props`接收到的数据，也可以传给`setup`使用，对`ChildComp`组件进行简单改造，代码如下所示：

```
<template>
  <div>原始编号{{ id }}，新的编号：{{ idNew }} ，真实姓名：{{ realName }}</div>
</template>

<script>
import { reactive, toRefs } from 'vue'
export default {
  props: ['id', 'realName'],
  setup(props) {
    const status = reactive({
      idNew: `0001${props.id}`
    })
    return { ...toRefs(status) }
  }
}
</script>
```

可以看到，把`props`作为参数传给`setup`函数，`setup`中就可以使用了。如果`status`中的属性和`props`重名，则以`props`中的优先。

（5）运行后，界面如下图所示：




（6）前面提到过，`props`也可以是对象

利用对象，可以给传递的参数指定类型，如上例中，用对象表达如下：

```
props: {
    id: Number,
    realName: String
}
```

这样有一个好处就是：它不仅为组件提供了文档，还会在遇到类型错误时，在浏览器的`JavaScript`控制台出现提示。

还有更高级的（了解即可），如下：

```
  props: {
    // 1.基础的类型检查
    propA: Number,
    // 2.多个可能的类型
    propB: [String, Number],
    // 3.必填的字符串
    propC: {
      type: String,
      required: true
    },
    // 4.带有默认值的数字
    propD: {
      type: Number,
      default: 100
    },
    // 5.带有默认值的对象
    propE: {
      type: Object,
      // 对象或数组默认值，必须从一个工厂函数获取
```

```
      default: function () {
        return { message: 'hello' }
      }
    },
    // 6.自定义验证函数
    propF: {
      validator: function (value) {
        // 这个值必须匹配下列字符串中的一个
        return ['success', 'warning', 'danger'].indexOf(value)
!== -1
      }
    }
  }
```

## 3.2 子组件传递数据给父组件

前面提到过，子组件传递数据给父组件，采用的方法是：`子组件通过自定义事件给父组件发生消息`。

主要步骤如下：

- 父组件编写一个方法给子组件调用，入口参数让子组件传过来，代码如下：

```
// 父组件setup方法中，定义一个方法，给子组件调用，childData即子组件传过来的数据
const getData = (childData) => {
    console.log(childData)
}
```

- 父组件通过`v-on(@)`绑定事件名（任意，这里用`fatherEvent`），把上述编写好的方法通过子组件标签传递给子组件调用，语法格式如下：

```
<ChildComp @fatherEvent="getData"></ChildComp>
```

- 子组件通过`setup`的`context对象`的`emit函数`，调用父组件传递过来的事件（`fatherEvent`），语法格式如下：

```
setup(props, context) {
    const state = reactive({
      count: 0
    })
    const checkMe = () => {
      context.emit('fatherEvent', '这是子组件传出的内容')
    }
    return { ...toRefs(state), checkMe }
  }
```

- 子组件什么时候给父组件传数据？这里写了一个按钮，点击按钮，就调用`fatherEvent`，给它传参数。

  完整代码如下：

  （1）`FatherComp.vue`：

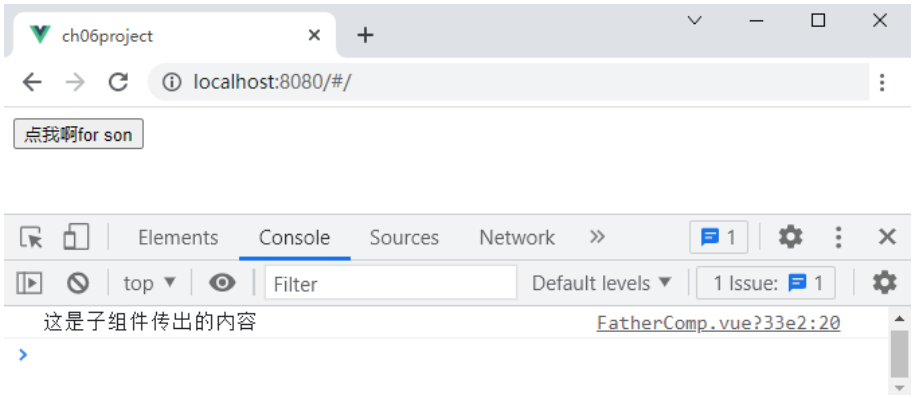
```
<template>
  <div>
    <!--3 使用子组件-->
    <ChildComp @fatherEvent="getData"></ChildComp>
  </div>
</template>

<script>
import { reactive, toRefs } from 'vue'
import ChildComp from './ChildComp.vue' //2.1.导入子组件
export default {
  components: {
    ChildComp //2.2.注册子组件
  },
  setup() {
    const state = reactive({
      count: 0
    })
    const getData = (childData) => {
      console.log(childData)
    }
```

```
    return {
      ...toRefs(state),
      getData
    }
  }
}
</script>
```

（2）`ChildComp.vue`：

```
<template>
  <div>
    <button @click="sendMsg">点我啊for son</button>
  </div>
</template>

<script>
import { reactive, toRefs } from 'vue'
export default {
  /*setup接收2个参数，第一个参数为props，它是一个对象。第二个参数context，它为非响应式对象。
    context对象包含三个属性：emit(方法)、slots(插槽对象)、attrs(attribute对象)，因此我们也可以对它进行解构使用，如：{emit}
  */
  setup(props, context) {
    const state = reactive({
      count: 0
    })
    const sendMsg = () => {
      context.emit('fatherEvent', '这是子组件传出的内容')
    }

    return { ...toRefs(state), sendMsg }
  }
}
</script>
```

运行结果，如下图所示：


ch06project
localhost:8080/#/
点我啊for son
Elements
Console
Sources
Network
top
Filter
Default levels
1 Issue: 1
这是子组件传出的内容
FatherComp.vue?33e2:20