# 监听子组件事件

向子组件传递方法还可以有另外一种方式，就是利用Vue提供的$emit方法.

$emit的功能比props要强大，可以用于监听自定义事件。

用法是

父组件中 @自定义事件="表达式或方法"

子组件中 @click="$emit('自定义事件')" 其中click可以换成其他事件

注意：若子组件的事件名中存在驼峰式命名如：`removeTarget` 父组件中就要写成 r`emove-target` 单词小写，通过`-`连接

## $emit用法,参数通过父组件传递

TodoList参考案例：

```
<body>
  <div id='todoapp'>
    <header>
      <h1>小海记事</h1>
      <input type="text" placeholder="请输入任务" v-model.trim="task"
@keyup.enter="add(task)">
    </header>
    <div class="main">
      <ul class="todo-list">
        <todo v-for="(item,index) in todo_list"
        :item="item" :index="index"
        @remove-target="remove(index)"></todo>
      </ul>
    </div>
    <div class="footer" v-show="todo_list.length != 0">
      <span class="todo-count"><strong>{{todo_list.length}} tasks left</strong>
</span>
      <button class="clear-completed" @click="empty">清空</button>
    </div>
  </div>

  <script src="../vue.js"></script>
  <script>

    //创建vue对象
    const app = Vue.createApp({
      data() {
        todo_list = ['吃饭', '睡觉', '上班', '学习']
        task = ''
        return {
          todo_list,
          task
        }
      },
      methods: {
        add(task) {
          if (task) {
```

```
            this.todo_list.push(task)  // 将任务保持到任务列表
            this.task = ''               // 清空文本框
          }
        },
        empty() {
          this.todo_list = []
        },
        remove(index) {
          console.log('remove',index)
          this.todo_list.splice(index,1)
        }
      }
    })
    //定义子组件
    const todo = {
      props: ['item', 'index'],
      template: `
      <li class="todo">
        <div class="view">
          <input type="checkbox" :id="index" :value="item">
          <span class="index">{{index+1}}. </span>
          <label :for="index">{{item}}</label>
          <button class="destory" @click="$emit('removeTarget')">X</button>
        </div>
      </li>
      `
    }
    //注册组件
    app.component('todo', todo)
    const vm = app.mount('#todoapp') // 返回组件实例
  </script>
</body>
```

这个例子中采用了 $emit方案代替props，代码比上一种更简洁，且复用性更高。

为什么呢？假设我在父组件中不断增加方法，那么子组件要用的话，如果采用props就必须传递过来才能用。如果采用emit,那么只需要在父组件中的自定义事件替换执行代码就可以。

## $emit用法,参数通过子组件传递

如果有的时候，每个子组件希望传递不一样的参数，这个时候需要将参数从父组件传递移动到子组件

此时的用法是：

父组件：@自定义事件="表达式或方法"

子组件：@click="$emit('自定义事件',参数)"　其中click可以换成其他事件

```
<body>
  <div id='todoapp'>
    <header>
      <h1>小海记事</h1>
      <input type="text" placeholder="请输入任务" v-model.trim="task"
@keyup.enter="add(task)">
    </header>
    <div class="main">
      <ul class="todo-list">
        <todo v-for="(item,index) in todo_list"
```

```
        :item="item" :index="index"
        @remove-target="remove"></todo>
      </ul>
    </div>
    <div class="footer" v-show="todo_list.length != 0">
      <span class="todo-count"><strong>{{todo_list.length}} tasks left</strong>
</span>
      <button class="clear-completed" @click="empty">清空</button>
    </div>
  </div>

  <script src="../vue.js"></script>
  <script>

    //创建vue对象
    const app = Vue.createApp({
      data() {
        todo_list = ['吃饭', '睡觉', '上班', '学习']
        task = ''
        return {
          todo_list,
          task
        }
      },
      methods: {
        add(task) {
          if (task) {
            this.todo_list.push(task)  // 将任务保持到任务列表
            this.task = ''                // 清空文本框
          }
        },
        empty() {
          this.todo_list = []
        },
        remove(index) {
          console.log('remove',index)
          this.todo_list.splice(index,1)
        }
      }
    })
    //定义子组件
    const todo = {
      props: ['item', 'index'],
      template: `
      <li class="todo">
        <div class="view">
          <input type="checkbox" :id="index" :value="item">
          <span class="index">{{index+1}}. </span>
          <label :for="index">{{item}}</label>
          <button class="destory"
@click="$emit('removeTarget',index)">X</button>
        </div>
      </li>
      `

    }
    //注册组件
    app.component('todo', todo)
    const vm = app.mount('#todoapp') // 返回组件实例
```

```
  </script>
</body>
```

## 组件上使用v-model

在自定义组件中使用双向数据绑定时需要进行额外处理，如果只是使用传统的方式会发现不生效。

```
<body>
  <div id="root">
    <search-input v-model="kwargs"></search-input>
    <h3>你输入的关键字是:{{kwargs}}</h3>
  </div>
  <script src="../vue.js"></script>
  <script>
    const app = Vue.createApp({
      data(){
        return{
          kwargs: ''
        }
      },
    })
    app.component('searchInput',{

      template: `
      <input type="text" placeholder="请输入搜索关键字">
      `
    })
    app.mount('#root')
  </script>
</body>
```

对此我们可以拆解下v-model在input中的实现机制

```
<input type="text" v-model="searchText">
等价于
<input type="text" :value="searchText" @input="searchText=$event.target.value">
```

可以通过此案例，发现是一样的效果

```
<body>
  <div id="root">
    <!-- <input type="text" v-model="searchText"> -->
    <input type="text" :value="searchText"
@input="searchText=$event.target.value">
    <h3>输入内容:{{searchText}}</h3>
  </div>
  <script src="../vue.js"></script>
  <script>
    Vue.createApp({
      data(){
        return{
          searchText:'',
        }
      }
```

```
    }).mount('#root')
  </script>
</body>
```

放在组件中，v-model的效果则变成

```
<custom-input
  :model-value="searchText"
  @update:model-value="searchText = $event"
></custom-input>
```

为了让它正常工作，这个组件内的 `<input>` 必须：

- 将其 `value` 属性绑定到一个名叫 `modelValue` 的 prop 上
- 在其 `input` 事件被触发时，将新的值通过自定义的 `update:modelValue` 事件抛出

最终，改造后v-model可以完整的工作起来了

```
<body>
  <div id="root">
    <search-input v-model="kwargs"></search-input>
    <h3>你输入的关键字是:{{kwargs}}</h3>
  </div>
  <script src="../vue.js"></script>
  <script>
    const app = Vue.createApp({
      data(){
        return{
          kwargs: ''
        }
      },
    })
    app.component('searchInput',{
      props:['modelValue'],
      emits:['update:modelValue'],
      template: `
      <input type="text" placeholder="请输入搜索关键字"
        :value="modelValue"
        @input="$emit('update:modelValue',$event.target.value)"
      >
      `
    })
    app.mount('#root')
  </script>
</body>
```



# 组件的插槽

目前为止，我们知道如何给组件传递数据 （props），传递事件（emits ） 。

但是如果要传递一组html元素给组件，我们可以采用插槽技术。

语法是在组件内通过`<slot></slot>`标签留下一组插槽，在父组件中使用的时候，可以用任何HTML内容注入此插槽。

```
<body>
  <div id="root">
    <alert-box>
      <h3>服务器错误</h3>
    </alert-box>
  </div>
  <script src="../vue.js"></script>
  <script>
    const app = Vue.createApp({})
    app.component('alert-box',{
      template: `
      <div class="demo-alert-box">
        <strong>Error!</strong>
        <slot></slot>
      </div>
      `
    })
    app.mount('#root')
  </script>
</body>
```

## Vue生命周期

每个组件在被创建时都要经过一系列的初始化过程——例如，需要设置数据监听、编译模板、将实例挂载到 DOM 并在数据变化时更新 DOM 等。同时在这个过程中也会运行一些叫做**生命周期钩子**的函数，这给了用户在不同阶段添加自己的代码的机会。

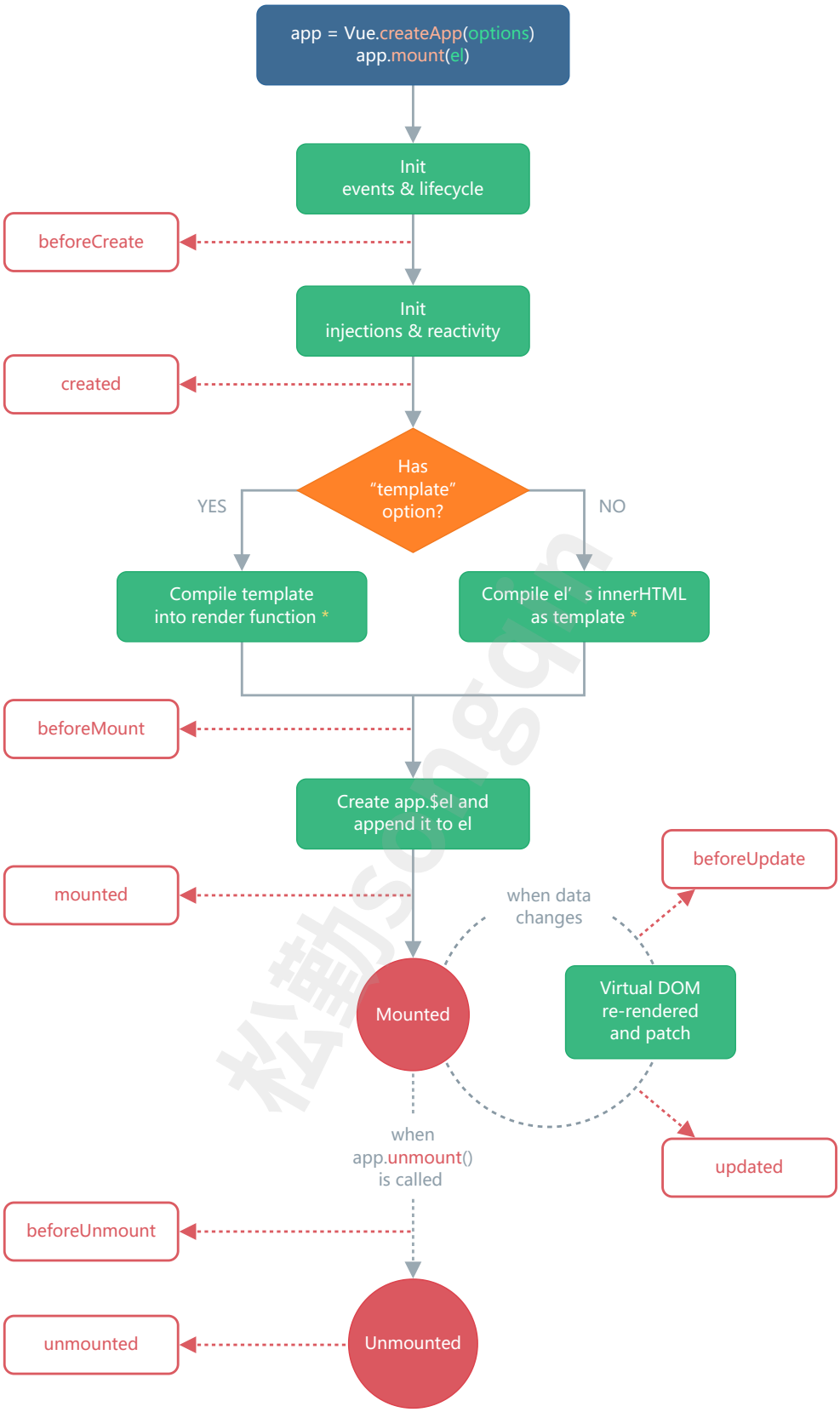
比如 created 钩子可以用来在一个实例被创建之后执行代码：

```
Vue.createApp({
  data() {
    return { count: 1}
  },
  created() {
    // `this` 指向 vm 实例
    console.log('count is: ' + this.count) // => "count is: 1"
  }
})
```

也有一些其它的钩子，在实例生命周期的不同阶段被调用，如 mounted、updated 和 unmounted。生命周期钩子的 `this` 上下文指向调用它的当前活动实例。

注意：不要用箭头函数来定义周期函数。比如 `created: () => console.log(this.a)`

下图展示了实例的生命周期。我们不需要立马弄明白所有的东西，不过随着不断学习和使用，它的参考价值会越来越高。




* Template compilation is performed ahead-of-time if using a build step, e.g., with single-file components.

这里可以通过一个案例演示从create到unmouted的过程,至于update和beforeupdate需要更新组件才能看见。

```
<body>
  <div id="root">
    <button @click='ate=!ate'>吃饭/消化</button>
    <life-demo v-if="ate">
      <h3>饿了么</h3>
    </life-demo>
  </div>
  <script src="../vue.js"></script>
  <script>
    const app = Vue.createApp({

      data(){
        return{
          ate: false
        }
      },
    })
    const comp = app.component('life-demo',{
      //生命周期钩子
      beforeCreate(){
        console.log('beforeCreated')
      },
      created(){
        console.log('created')
      },
      beforeMount(){
        console.log('beforeMount')
      },
      mounted(){
        console.log('mounted')
      },
      beforeUnmount(){
        console.log('beforeUnmount')
      },
      unmounted(){
        console.log('unmounted')
      },
      template:`
      <div>
        <slot></slot>
      </div>
      `
    })
    const vm = app.mount('#root')
  </script>
</body>
```



更新组件可以通过更新数据查看,点击更新链接或更新背景都会触发DOM的重新渲染从而触发updated和beforeupdate两个生命周期钩子。

```
<!DOCTYPE html>
<html lang="en">
<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>demo6</title>
  <style>
    .green{
      background-color: green;
    }
    .red{
      background-color: red;
    }
  </style>
</head>
<body>
  <div id="root">
    <button @click="changeLink">更新链接</button>
    <button @click="changeColor">更新背景</button>
    <ul>
      <blog-post v-for="link in links" :item="item" :link="link">
      </blog-post>
    </ul>
  </div>
  <script src="../vue.js"></script>
  <script>
    const app = Vue.createApp({
      data(){
        return{
          links:[
            'baidu.com',
            'qq.com',
            'taobao.com',
          ],
          item:'red'
        }
      },
      methods:{
        changeLink(){
          this.links=[
            'aiqiyi.com',
            'meituan.com',
            'jingdong.com',
          ]
        },
        changeColor(){
          this.item=this.item=='green'?'red':'green'
        }
      }
    })
    app.component('blog-post',{
      beforeUpdate(){
        console.log('beforeUpdate')
      },
      updated(){
        console.log('update')
      },
      props:['item','link'],
      template:`
        <li :class="item"><a :href="link">{{link}}</a></li>
      `
    })
```

```
    app.mount('#root')
  </script>
</body>
```

至此主要生命周期钩子函数介绍完毕，在实际使用中，可以在钩子函数里加些额外方法来控制逻辑。后面项目实战再具体介绍。