

# 组件化 – 额外知识补充

王红元 coderwhy

# 目录

## content



**1 组件的生命周期**

**2 组件中的ref引用**

**3 动态组件的使用**

**4 keep-alive组件**

**5 异步组件的使用**

**6 组件的v-model**

**7 组件的混入Mixin**

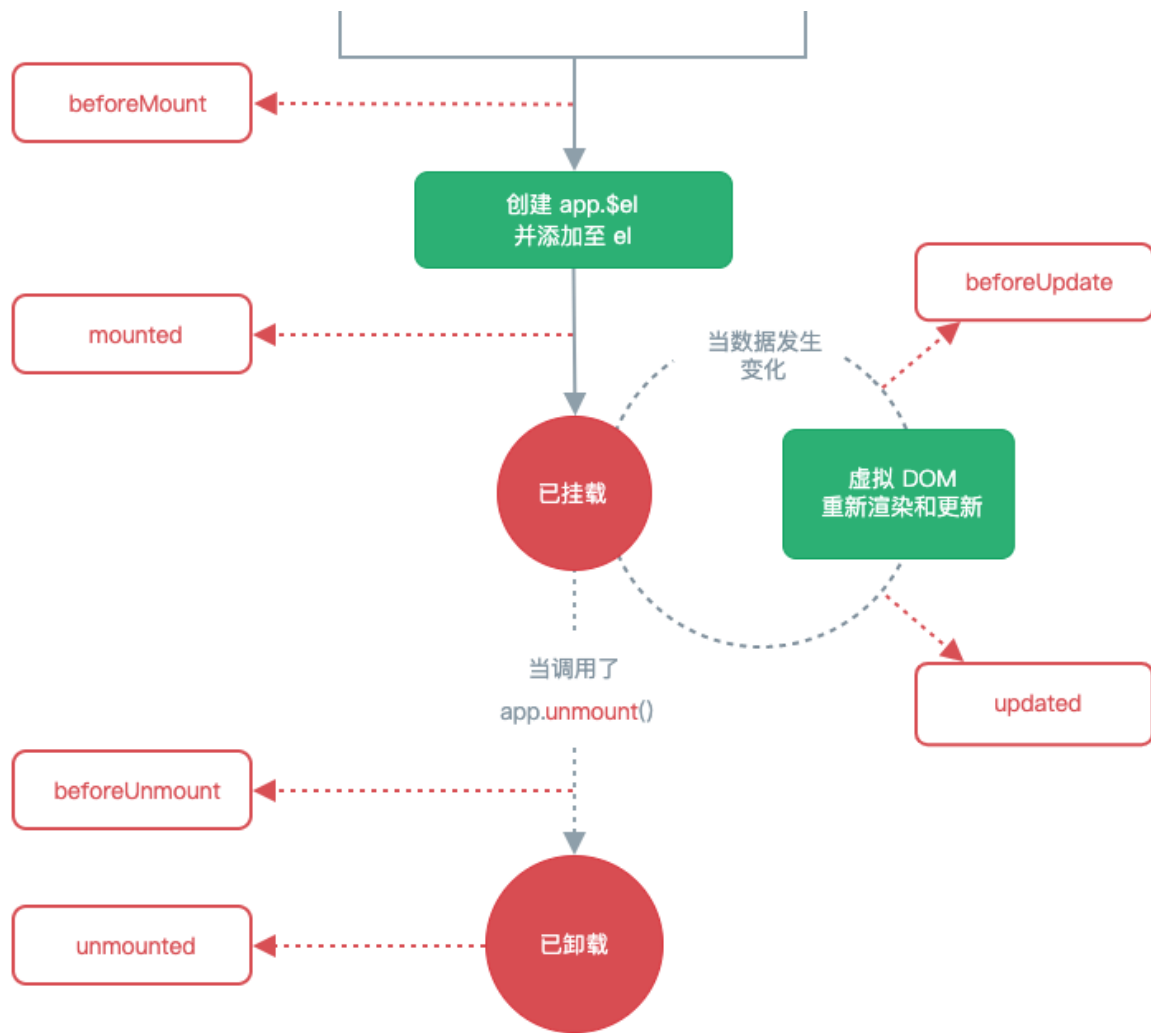
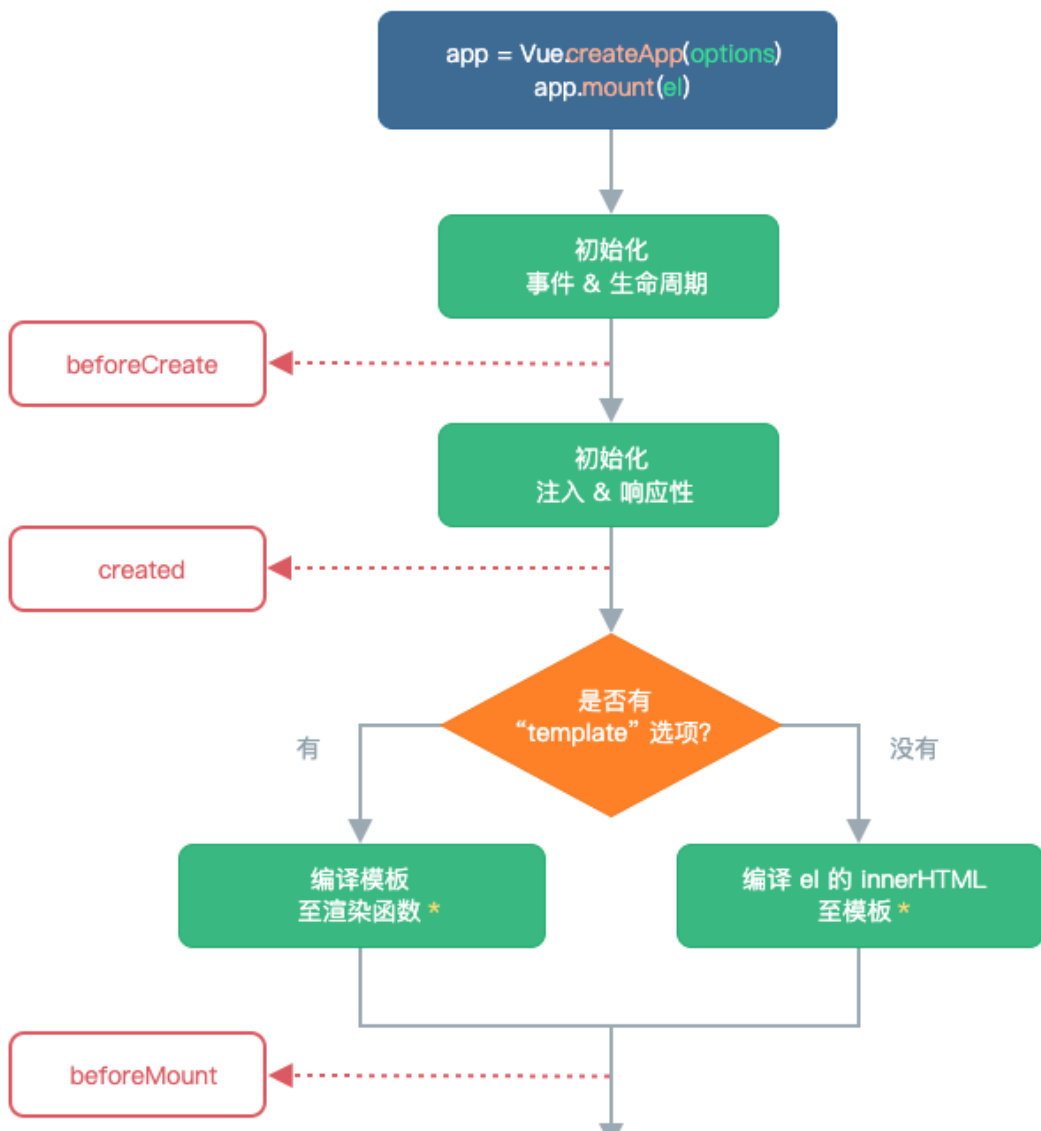
## ■ 什么是生命周期呢？

- 生物学上，生物生命周期指得是一个生物体在生命开始到结束周而复始所历经的一系列变化过程；
- 每个组件都可能会经历从创建、挂载、更新、卸载等一系列的过程；
- 在这个过程中的某一个阶段，我们可能会想要添加一些属于自己的代码逻辑（比如组件创建完后就请求一些服务器数据）；
- 但是我们如何可以知道目前组件正在哪一个过程呢？Vue给我们提供了组件的生命周期函数；

## ■ 生命周期函数：

- 生命周期函数是一些钩子函数（回调函数），在某个时间会被Vue源码内部进行回调；
- 通过对生命周期函数的回调，我们可以知道目前组件正在经历什么阶段；
- 那么我们就可以在该生命周期中编写属于自己的逻辑代码了；

# 生命周期的流程



# \$refs的使用

## ■ 某些情况下，我们在组件中想要直接获取到元素对象或者子组件实例：

- 在Vue开发中我们是不推荐进行DOM操作的；
- 这个时候，我们可以给元素或者组件绑定一个ref的attribute属性；

## ■ 组件实例有一个\$refs属性：

- 它是一个对象Object，持有注册过 ref attribute 的所有 DOM 元素和组件实例。

```
visitElement() {  
  // 访问元素  
  console.log(this.$refs.title);  
  // 访问组件实例  
  console.log(this.$refs.helloCpn.$el);  
  // 访问组件实例  
  this.$refs.helloCpn.showMessage();  
}
```

# \$parent和\$root

- 我们可以通过\$parent来访问父元素。

- HelloWorld.vue的实现：

- 这里我们也可以通过\$root来实现，因为App是我们的根组件；

```
visitParent() {  
  console.log(this.$parent.message);  
  console.log(this.$root.message);  
}
```

- 注意：在Vue3中已经移除了\$children的属性，所以不可以使用了。

# 切换组件案例

## ■ 比如我们现在想要实现了一个功能：

- 点击一个tab-bar，切换不同的组件显示；

home about category

Home组件

## ■ 这个案例我们可以通过两种不同的实现思路来实现：

- 方式一：通过v-if来判断，显示不同的组件；
- 方式二：动态组件的方式；

# v-if显示不同的组件

- 我们可以先通过v-if来判断显示不同的组件，这个可以使用我们之前讲过的知识来实现：

```
<template>
  <div>
    <button v-for="tab in tabs"
      :key="tab"
      :class="{active: currentTab === tab}"
      @click="tabClick(tab)">
      {{tab}}
    </button>
    <template v-if="currentTab === 'home'">
      <home></home>
    </template>
    <template v-else-if="currentTab === 'about'">
      <about></about>
    </template>
    <template v-else>
      <category></category>
    </template>
  </div>
</template>
```

made by coderwhy



# 动态组件的实现

- 动态组件是使用 **component 组件**，通过一个**特殊的attribute is** 来实现：

```
<template>
  <div>
    <button v-for="tab in tabs"
      :key="tab"
      :class="{active: currentTab === tab}"
      @click="tabClick(tab)">
      {{tab}}
    </button>

    <component :is="currentTab"></component>
  </div>
</template>
```

- 这个currentTab的值需要是什么内容呢？

- 全局注册：可以是通过**component函数注册**的组件；
- 布局注册：在一个**组件对象的components对象中注册**的组件；

# 动态组件的传值

## ■ 如果是动态组件我们可以给它们传值和监听事件吗？

- 也是一样的；
- 只是我们需要将属性和监听事件放到component上来使用；

```
<component name="why"
  :age="18"
  @pageClick="pageClick"
  :is="currentTab"/>
```

# 认识keep-alive

## ■ 我们先对之前的案例中About组件进行改造：

- 在其中增加了一个按钮，点击可以递增的功能；

home about category  
About组件 0

## ■ 比如我们将counter点到10，那么在切换到home再切换回来about时，状态是否可以保持呢？

- 答案是否定的；

- 这是因为默认情况下，我们在切换组件后，about组件会被销毁掉，再次回来时会重新创建组件；

## ■ 但是，在开发中某些情况我们希望继续保持组件的状态，而不是销毁掉，这个时候我们就可以使用一个内置组件：keep-alive。

```
<keep-alive include="home,about">
  <component name="why"
    :age="18"
    @pageClick="pageClick"
    :is="currentTab"/>
</keep-alive>
```

# keep-alive属性

## ■ keep-alive有一些属性:

- ❑ **include** - string | RegExp | Array。只有名称匹配的组件会被缓存;
- ❑ **exclude** - string | RegExp | Array。任何名称匹配的组件都不会被缓存;
- ❑ **max** - number | string。最多可以缓存多少组件实例,一旦达到这个数字,那么缓存组件中最近没有被访问的实例会被销毁;

## ■ include 和 exclude prop 允许组件有条件地缓存:

- ❑ 二者都可以用逗号分隔字符串、正则表达式或一个数组来表示;
- ❑ 匹配首先检查组件自身的 **name** 选项;

```
<!-- 逗号分隔字符串 -->
<keep-alive include="a,b">
  <component :is="view"></component>
</keep-alive>

<!-- regex (使用 `v-bind`) -->
<keep-alive :include="/a|b/">
  <component :is="view"></component>
</keep-alive>

<!-- Array (使用 `v-bind`) -->
<keep-alive :include="['a','b']">
  <component :is="view"></component>
</keep-alive>
```

# 缓存组件的生命周期

- 对于缓存的组件来说，再次进入时，我们是**不会执行created或者mounted等生命周期函数**的：
  - 但是有时候我们确实希望监听到何时重新进入到了组件，何时离开了组件；
  - 这个时候我们可以使用**activated** 和 **deactivated** 这两个生命周期钩子函数来监听；

```
activated() {  
  console.log("about activated")  
},  
deactivated() {  
  console.log("about deactivated")  
}
```

# Webpack的代码分包

## ■ 默认的打包过程：

- 默认情况下，在构建整个组件树的过程中，因为组件和组件之间是**通过模块化直接依赖**的，那么**webpack在打包时就会将组件模块打包到一起**（比如一个app.js文件中）；
- 这个时候随着**项目的不断庞大**，**app.js文件的内容过大**，会造成**首屏的渲染速度变慢**；

## ■ 打包时，代码的分包：

- 所以，对于一些**不需要立即使用的组件**，我们可以**单独对它们进行拆分**，拆分成一些**小的代码块chunk.js**；
- 这些chunk.js会在需要时**从服务器加载下来**，并且**运行代码**，显示对应的内容；

## ■ 那么webpack中如何可以对代码进行分包呢？



# Vue中实现异步组件

- 如果我们的项目过大了，对于**某些组件**我们希望通过**异步的方式来进行加载**（目的是可以对其进行分包处理），那么Vue中给我们提供了一个函数：**defineAsyncComponent**。
- **defineAsyncComponent**接受两种类型的参数：
  - **类型一**：工厂函数，该工厂函数需要返回一个Promise对象；
  - **类型二**：接受一个对象类型，对异步函数进行配置；
- **工厂函数类型一的写法**：

```
<script>
import { defineAsyncComponent } from 'vue';
const AsyncHome = defineAsyncComponent(() => import("./AsyncHome.vue"));

export default {
  components: {
    AsyncHome
  }
}
</script>
```

# 异步组件的写法二（了解）

```
const AsyncHome = defineAsyncComponent({
  // 工厂函数
  loader: () => import("./AsyncHome.vue"),
  // 加载过程中显示的组件
  loadingComponent: Loading,
  // 加载失败时显示的组件
  errorComponent: Error,
  // 在显示 loadingComponent 之前的延迟 | 默认值: 200 (单位 ms)
  delay: 2000,
  // 如果提供了 timeout, 并且加载组件的时间超过了设定值, 将显示错误组件
  // 默认值: Infinity (即永不超时, 单位 ms)
  // timeout: 0,
  // 定义组件是否可挂起 | 默认值: true
  suspense: true
});
```



# 组件的v-model

## ■ 前面我们在input中可以使用v-model来完成双向绑定：

□ 这个时候往往会非常方便，因为v-model默认帮助我们完成了两件事；

□ v-bind:value的数据绑定和@input的事件监听；

## ■ 如果我们现在封装了一个组件，其他地方在使用这个组件时，是否也可以使用v-model来同时完成这两个功能呢？

□ 也是可以的，vue也支持在组件上使用v-model；

## ■ 当我们在组件上使用的时候，等价于如下的操作：

□ 我们会发现和input元素不同的只是属性的名称和事件触发的名称而已；

```
<my-input v-model="message"/>
```

```
<!-- 相当于 -->
```

```
<my-input :model-value="message" @update:model-value="message = $event"></my-input>
```

# 组件v-model的实现

- 那么，为了我们的MyInput组件可以正常的工作，这个组件内的 `<input>` 必须：
  - 将其 value attribute 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；
- MyInput.vue的组件代码如下：

```
<template>
  <div>
    <input :value="modelValue" @input="inputChange">
  </div>
</template>

<script>
  export default {
    props: ["modelValue"],
    emits: ["update:modelValue"],
    methods: {
      inputChange(event) {
        this.$emit("update:modelValue", event.target.value);
      }
    }
  }
</script>
```

```
<my-input v-model="message"/>
```

# 绑定多个属性

- 我们现在通过v-model是直接绑定了一个属性，如果我们希望绑定多个属性呢？
  - 也就是我们希望在~~一个组件上使用多个v-model~~是否可以实现呢？
  - 我们知道，默认情况下的v-model其实是绑定了 modelValue 属性和 @update:modelValue的事件；
  - 如果我们希望绑定更多，可以给v-model传入一个参数，那么这个参数的名称就是我们绑定属性的名称；

- 注意：这里我是绑定了两个属性的

```
<my-input v-model="message" v-model:title="title"/>
```

- v-model:title相当于做了两件事：

- 绑定了title属性；
- 监听了 @update:title的事件；

```
export default {
  props: ["modelValue", "title"],
  emits: ["update:modelValue", "update:title"],
  methods: {
    input1Change(event) {
      this.$emit("update:modelValue", event.target.value);
    },
    input2Change(event) {
      this.$emit("update:title", event.target.value);
    }
  }
}
```

- 目前我们是使用组件化的方式在开发整个Vue的应用程序，但是**组件和组件之间有时候会存在相同的代码逻辑**，我们希望对**相同的代码逻辑进行抽取**。
- 在Vue2和Vue3中都支持的一种方式就是**使用Mixin来完成**：
  - Mixin提供了一种非常灵活的方式，来**分发Vue组件中的可复用功能**；
  - 一个Mixin对象可以包含**任何组件选项**；
  - 当组件使用Mixin对象时，所有**Mixin对象的选项将被 混合 进入该组件本身的选项**中；

# Mixin的基本使用

V Home.vue U X

src &gt; 14\_Mixin混入 &gt; pages &gt; V Home.vue &gt; {} "Home.vue"

```
1 <template>
2   <div>
3     <button @click="foo">foo点击</button>
4   </div>
5 </template>
6
7 <script>
8   import sayHelloMixin from '../mixins/sayHello';
9
10  export default {
11    mixins: [sayHelloMixin]
12  }
13 </script>
14
15 <style scoped>
16
17 </style>
```

JS sayHello.js U X

src &gt; 14\_Mixin混入 &gt; mixins &gt; JS sayHello.js &gt; [E] default

```
1 const sayHelloMixin = {
2   created() {
3     this.sayHello();
4   },
5   methods: {
6     sayHello() {
7       console.log("Hello Page Component");
8     }
9   }
10 }
11
12 export default sayHelloMixin;
```

# Mixin的合并规则

## ■ 如果Mixin对象中的选项和组件对象中的选项发生了冲突，那么Vue会如何操作呢？

- 这里分成不同的情况来进行处理；

## ■ 情况一：如果是data函数的返回值对象

- 返回值对象默认情况下会进行合并；

- 如果data返回值对象的属性发生了冲突，那么会保留组件自身的数据；

## ■ 情况二：如何生命周期钩子函数

- 生命周期的钩子函数会被合并到数组中，都会被调用；

## ■ 情况三：值为对象的选项，例如 methods、components 和 directives，将被合并为同一个对象。

- 比如都有methods选项，并且都定义了方法，那么它们都会生效；

- 但是如果对象的key相同，那么会取组件对象的键值对；

# 全局混入Mixin

■ 如果组件中的某些选项，是所有的组件都需要拥有的，那么这个时候我们可以使用**全局的mixin**：

- 全局的Mixin可以使用 **应用app的方法 mixin** 来完成注册；
- 一旦注册，那么**全局混入的选项将会影响每一个组件**；

```
const app = createApp(App);
app.mixin({
  created() {
    console.log("global mixin created");
  }
})
app.mount("#app");
```