vue2和vue3基础

vue.js是什么

- 是一套用于构建用户界面的渐进式框架
- Vue 被设计为可以自底向上逐层应用
- Vue 的核心库只关注视图层
  demo:

```html
<!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">
    <script src="https://cdn.jsdelivr.net/npm/vue@2/dist/vue.js"></script>
    <title>Document</title>
</head>
<body>
    <div id="app">
        {{ message }}
      </div>
      <script>
        var app = new Vue({
        el: '#app',
        data: {
          message: 'Hello Vue!'
        }
      })
```

```
25          </script>

    </body>
    </html>
```

从0开始搭建一个vue环境

vue开发环境很多，这里新手从基本的开始，这样比较好理解，后面那些脚手架其实原理也是一样。

环境安装

- 安装node
- npm i vue
- 安装 npm i -D webpack webpack-cli webpack-dev-server
- 安装 pm i -D babel-loader @babel/core @babel/preset-env
- 安装 npm i -D vue-loader css-loader vue-style-loader vue-template-compiler html-webpack-plugin
- 目录
  - node_modules 模块包
  - src 代码目录
  - .babelrc.js babel降级适配
  - jsconfig.json 表示该目录是JavaScript项目的根目录,根文件和JavaScript语言服务提供的功能选项。
  - package-lock.json 锁定版本号,各种依赖包信息
  - package.json 记录你项目中所需要的所有模块
  - webpack.config.js 打包工具配置

基本语法

插值表达式

dom中插值表达式赋值, vue的变量必须在data里声明

`语法：{{ 表达式 }}`

- `{{ msg }}`

- `{{ obj.name }}`
- `{{ obj.age > 18 ? '成年' : '未成年' }}`

v-text和v-html

更新DOM对象的innerText/innerHTML

- 语法:
  - v-text="vue数据变量"
  - v-html="vue数据变量"

v-bind

把vue变量的值,赋予给dom属性上,影响标签显示效果，注意这里是单向的,vue数据 -> dom

- 语法：`v-bind:属性名="vue变量"`
- 简写：`:属性名="vue变量"`

v-model

双向数据绑定

- 数据变化 -> 视图自动同步
- 视图变化 -> 数据自动同步

语法: v-model="vue数据变量"

高阶语法：
v-model.修饰符="vue数据变量"

- .number 以parseFloat转成数字类型
- .trim 除首尾空白字符
- .lazy 在change时触发而非inupt时

v-on

给标签绑定事件

- v-on:事件名="要执行的少量代码"
- v-on:事件名="methods中的函数"
- v-on:事件名="methods中的函数(实参)"
- @事件名.修饰符="methods里函数"
  - .stop - 阻止事件冒泡
  - .prevent - 阻止默认行为
  - .once - 程序运行期间, 只触发一次事件处理函数
- @keyup.enter - 监测回车按键
- @keyup.esc - 监测返回按键
- more： https://v2.cn.vuejs.org/v2/guide/events.html

v-for

列表渲染, 所在标签结构, 按照数据数量, 循环生成。可以遍历数组 / 对象 / 数字 / 字符串 (可遍历结构)

- 语法
  - v-for="(值, 索引) in 目标结构"
  - v-for="值 in 目标结构"

```
<div v-for="(value, key) in tObj" v-bind:key="value">
```

- key属性跟踪和更新元素

v-for更新监测

数组变更方法, 就会导致v-for更新, 页面更新

数组非变更方法, 返回新数组, 就不会导致v-for更新, 可采用覆盖数组或 `this.$set()`。vue不能监测到数组里赋值的动作而更新，如果需要请使用 `Vue.set()` 或者`this.$set()`, 或者覆盖整个数

v-show

v-show：控制标签的隐藏或出现

v-show：通过css控制样式style达成显示、隐藏效果。v-show 用的display:none 隐藏 (频繁切换使用))

语法：
v-show="vue变量"

v-show有更高的渲染消耗，如果需要频繁切换，则v-show 较合适。

v-if

v-if：条件判断
经常和v-else-if,v-else一起使用

v-if：通过 创建、销毁 dom元素的方式达到显示、隐藏效果(销毁后有一个占位符)。

v-if 有更高的切换消耗 ，如果运行条件不大可能改变，则v-if 较合适。

v-if使得元素被隐藏后，这个元素的物理位置有一个名称为的占位符，其与html的注释信息没有关系。

vue过滤器

过滤器只能用在, 插值表达式和v-bind表达式
语法：

- `{{ 值 | 过滤器名字 }}`

- ```
  // 只能在当前vue文件内使用
    data里面语法：
    filters: {
      过滤器名字 (val) {
        return 处理后的值
      }
    }
  ```

- 过滤器传参: vue变量 | 过滤器(实参)
- 多个过滤器: vue变量 | 过滤器1 | 过滤器2更

vue计算属性-computed

计算属性是基于它们的依赖项的值结果进行缓存的，只要依赖的变量不变, 都直接从缓存取结果
语法:
  ```

```
computed: {
    "计算属性名" () {
        return "值"
    }
}
  data里面语法:
  computed: {
    计算属性名 () {
      return 值
    }
  }

// 注意: 计算属性和data属性都是变量-不能重名
// 注意2: 函数内变量变化, 会自动重新计算结果返回
```

vue侦听器（watch）

可以侦听data/computed属性值改变

语法:

- ```
  watch: {
      "被侦听的属性名" (newVal, oldVal){

      }
  }
  ```

- ```
  watch: {
  "要侦听的属性名": {
      immediate: true, // 立即执行
      deep: true, // 深度侦听复杂类型内变化
      handler (newVal, oldVal) {}
  }
  }
  ```

自定义指令

https://cn.vuejs.org/guide/reusability/custom-directives.html#directive-hooks

组件

每个组件都是一个独立的个体, 代码里体现为一个独立的.vue文件

哪部分标签复用, 就把哪部分封装到组件内
(重要): 组件内template只能有一个根标签
(重要): 组件内data必须是一个函数, 独立作用域

重要的思想就是拆组件，提高代码的复用性，便捷性等

全局 - 注册使用

全局入口在main.js, 在new Vue之上注册

语法:

```
import Vue from 'vue'
import 组件对象 from 'vue文件路径'

Vue.component("组件名", 组件对象)
```

main.js - 立即演示

```
// 目标: 全局注册 (一处定义到处使用)
// 1. 创建组件 - 文件名.vue
// 2. 引入组件
import Pannel from './components/Pannel'
// 3. 全局 - 注册组件
/*
  语法:
  Vue.component("组件名", 组件对象)
```

```
*/
Vue.component("PannelG", Pannel)
```

全局注册PannelG组件名后，就可以当做标签在任意Vue文件中template里用

单双标签都可以或者小写加-形式，运行后，会把这个自定义标签当做组件解析，使用组件里封装的标签替换到这个位置

```
<PannelG></PannelG>
<PannelG/>
<pannel-g></pannel-g>
```

局部 - 注册使用

```
import 组件对象 from 'vue文件路径'

export default {
    components: {
        "组件名": 组件对象
    }
}
```

```
<template>
  <div id="app">
    <!-- 4. 组件名当做标签使用 -->
    <!-- <组件名></组件名> -->
    <PannelG></PannelG>
    <PannelL></PannelL>
  </div>
</template>

<script>
// 目标: 局部注册 (用的多)
// 1. 创建组件 - 文件名.vue
// 2. 引入组件
import Pannel from './components/Pannel_1'
```

```
export default {
  // 3. 局部 - 注册组件
  /*
    语法:
    components: {
      "组件名": 组件对象
    }
  */
  components: {
    PannelL: Pannel
  }
}
</script>
```

scoped

```
<style scoped>
```

在style上加入scoped属性, 就会在此组件的标签上加上一个随机生成的data-v开头的属性

而且必须是当前组件的元素, 才会有这个自定义属性, 才会被这个样式作用到

vue组件通信_子向父

语法:

- 父: @自定义事件名="父methods函数"
- 子: this.$emit("自定义事件名", 传值) - 执行父methods里函数代码

vue组件通信_父向子

> props变量本身是只读不能重新赋值，在子组件用props接受

使用 props这种方式把父组件的数据传给子组件，在调用组件时使用v-bind将参数赋给这个组件
目标：从父到子的数据流向,叫单向数据流

原因: 子组件修改, 不通知父级, 造成数据不一致性

如果第一个`MyProduct.vue`内自己修改商品价格为5.5, 但是`App.vue`里原来还记着18.8 - 数据 不一致了

所以: Vue规定props里的变量, 本身是只读的

vue组件通信-EventBus

两个组件的关系非常的复杂，通过父子组件通讯是非常麻烦的。这时候可以使用通用的组件通讯方案：事件总线（event-bus)

动态组件

场景: 同一个挂载点要切换不同组件显示

1. 创建要被切换的组件 - 标签+样式
2. 引入到要展示的vue文件内, 注册
3. 变量-承载要显示的组件名
4. 设置挂载点
5. 点击按钮-切换comName的值为要显示的组件名

组件缓存

组件切换会导致组件被频繁销毁和重新创建, 性能不高
使用Vue内置的keep-alive组件, 可以让包裹的组件保存在内存中不被销毁

```
<div style="border: 1px solid red;">
    <!-- Vue内置keep-alive组件, 把包起来的组件缓存起来 -->
    <keep-alive>
        <component :is="comName"></component>
    </keep-alive>
</div>
```

注意:

> 被缓存的组件不再创建和销毁, 而是激活和非激活

补充2个钩子方法名:

activated – 激活时触发

deactivated – 失去激活状态触发

组件插槽

用于实现组件的内容分发, 通过 slot 标签, 可以接收到写在组件标签内的内容
用法:

1. 组件内用`<slot></slot>`占位
2. 使用组件时夹着的地方, 传入标签替换slot

`v-bind可以省略成: v-on: 可以省略成@ 那么v-slot: 可以简化成#`

`<slot>夹着内容默认显示内容, 如果不给插槽slot传东西, 则使用<slot>夹着的内容在原地显示。slot的name属性起插槽名, 使用组件时, template配合#插槽名传入具体标签`

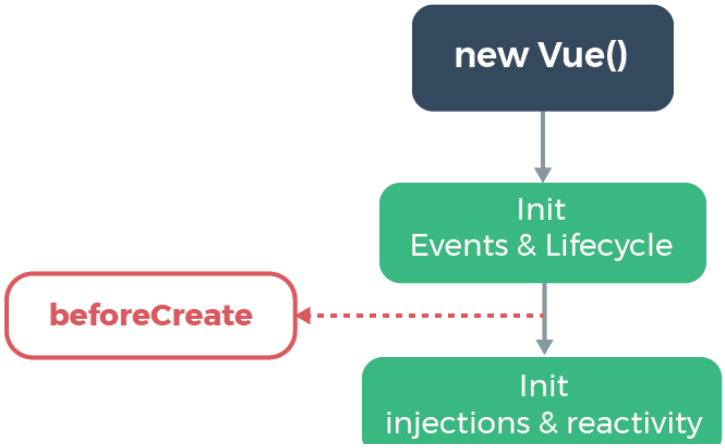
作用域插槽

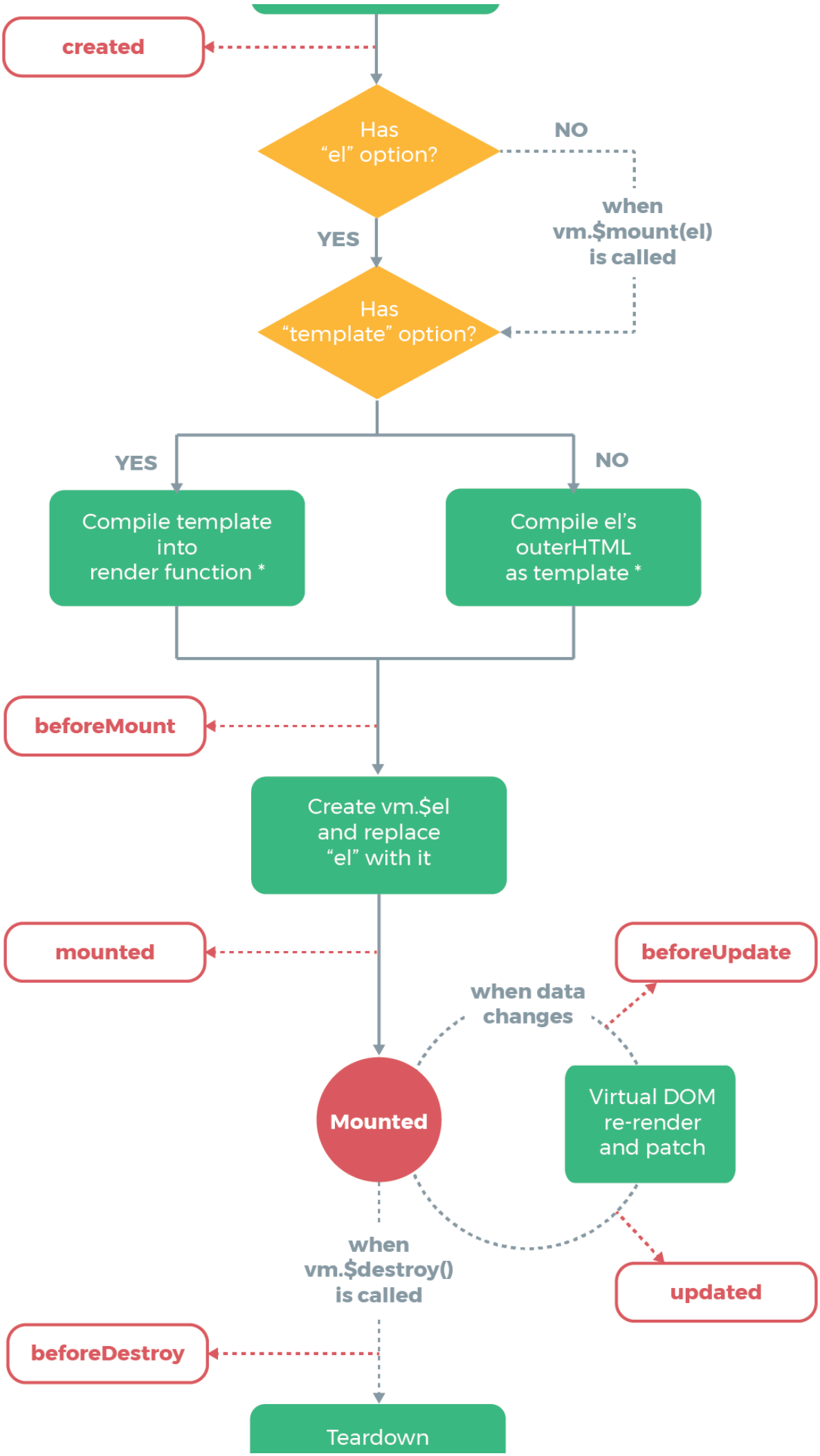
slot-scope的作用就是让父组件可以使用子组件data里面的数据
子组件里值, 在给插槽赋值时在父组件环境下使用

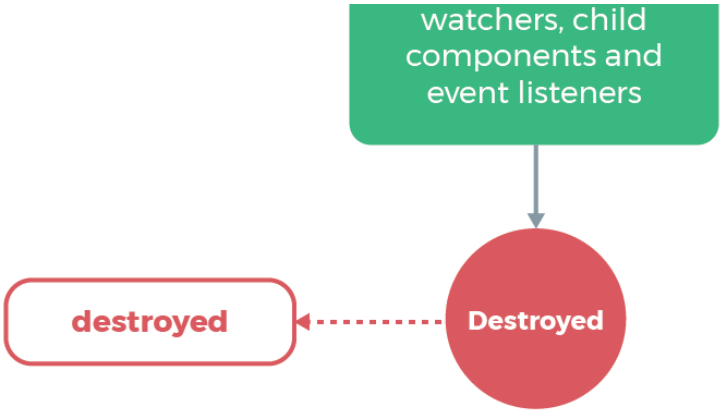
1. 子组件, 在slot上绑定属性和子组件内的值
2. 使用组件, 传入自定义标签, 用template和v-slot="自定义变量名"
3. scope变量名自动绑定slot上所有属性和值

vue生命周期

https://v2.cn.vuejs.org/v2/guide/instance.html#%E7%94%9F%E5%91%BD%E5%91%A8%E6%9C%9F%E5%9B%BE%E7%A4%BA




created
Has
“el” option?
NO
when
vm.$mount(el)
is called
YES
Has
“template” option?
YES
NO
Compile template
into
render function *
Compile el’s
outerHTML
as template *
beforeMount
Create vm.$el
and replace
“el” with it
mounted
beforeUpdate
when data
changes
Mounted
Virtual DOM
re-render
and patch
when
vm.$destroy()
is called
updated
beforeDestroy
Teardown



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

分类: 4大阶段8个方法

- 初始化
- 挂载
- 更新
- 销毁

阶段	方法名	方法名
初始化	beforeCreate	created
挂载	beforeMount	mounted
更新	beforeUpdate	updated
销毁	beforeDestroy	destroyed

初始化

1. new Vue() – Vue实例化(组件也是一个小的Vue实例)
2. Init Events & Lifecycle – 初始化事件和生命周期函数
3. beforeCreate – 生命周期钩子函数被执行
4. Init injections&reactivity – Vue内部添加data和methods等
5. created – 生命周期钩子函数被执行, 实例创建
6. 接下来是编译模板阶段 –开始分析

7. Has el option? – 是否有el选项 – 检查要挂到哪里
   -没有. 调用$mount()方法
   -有, 继续检查template选项

挂载

8. template选项检查
   有 - 编译template返回render渲染函数
   无 – 编译el选项对应标签作为template(要渲染的模板)
9. 虚拟DOM挂载成真实DOM之前
10. beforeMount – 生命周期钩子函数被执行
11. Create … – 把虚拟DOM和渲染的数据一并挂到真实DOM上
12. 真实DOM挂载完毕
13. mounted – 生命周期钩子函数被执行

更新阶段

14. 当data里数据改变, 更新DOM之前
15. beforeUpdate – 生命周期钩子函数被执行
16. Virtual DOM…… – 虚拟DOM重新渲染, 打补丁到真实DOM
17. updated – 生命周期钩子函数被执行
18. 当有data数据改变 – 重复这个循环

销毁阶段

19. 当$destroy()被调用 – 比如组件DOM被移除(例v-if)
20. beforeDestroy – 生命周期钩子函数被执行
21. 拆卸数据监视器、子组件和事件侦听器
22. 实例销毁后, 最后触发一个钩子函数
23. destroyed – 生命周期钩子函数被执行

axios

```
axios({
  method: '请求方式', // get post
  url: '请求地址',
  data: {    // 拼接到请求体的参数,  post请求的参数
    xxx: xxx,
  },
  params: {  // 拼接到请求行的参数, get请求的参数
            xxx: xxx
  }
}).then(res => {
  console.log(res.data) // 后台返回的结果
}).catch(err => {
  console.log(err) // 后台报错返回
})
```

避免前缀基地址, 暴露在逻辑页面里, 统一设置:

- axios.defaults.baseURL

$refs-获取DOM

ref定义值, 通过$refs.值 来获取组件对象, 就能继续调用组件内的变量

（1）在元素上 ref="自定义名称"
（2）在获取时 this.$refs.自定义名称

```
// 目标: 获取组件对象
// 1. 创建组件/引入组件/注册组件/使用组件
// 2. 组件起别名ref
// 3. 恰当时机, 获取组件对象
<template>
  <div>
    <h1 ref="myH1">1. ref获取原生dom</h1>
    <button @click="fn">点击修改上面内容</button>
  </div>
</template>
```

```
<script>
export default {
  methods: {
    fn() {
      console.log(this.$refs.myH1); // <h1></h1> 原生DOM标签
      this.$refs.myH1.innerHTML = "改内容了";
    }
  }
}
</script>
```

$nextTick使用

原因: Vue更新DOM异步

解决: this.$nextTick()过程：DOM更新完会挨个触发nextTick里的函数体

dom更新是异步的,直接获取内容,是拿不到更新后的值的,用$nextTick解决

```
<template>
  <div>
    <p id="num">数字: {{ count }}</p>
    <button @click="btn">点击+1, 观察打印</button>
  </div>
</template>

<script>
export default {
  data() {
    return {
      count: 0,
    };
  },
  methods: {
    btn() {
        this.count++; // 数字添加后, 异步更新DOM, 所以马上获取标签的值还是0
```

```
console.log(document.getElementById('num').innerHTML);
// 0
        this.$nextTick(() => {
            console.log("DOM更新后触发$nextTick函数");

console.log(document.getElementById('num').innerHTML);
// 1
        })
    },
  },
};
</script>
```

自定义指令

pass

elementUI-vue2.x

https://element.eleme.cn/#/zh-CN/component/installation

form属性

- rules 表单验证规则

vue路由

https://router.vuejs.org/zh/guide/#javascript

1.安装`yarn add vue-router`

2.`import VueRouter from 'vue-router'`

3.导入路由

```
// 在vue中，使用使用vue的插件，都需要调用Vue.use()
Vue.use(VueRouter)
```

4.创建路由规则数组

```
  {
    path: "/find",
    component: Find
  },
  {
    path: "/my",
    component: My
  },
  {
    path: "/part",
    component: Part
  }
]
```

5.创建路由对象 - 传入规则

```
const router = new VueRouter({
  routes
})
```

6.关联到vue实例

```
new Vue({
  router
})
```

7.

```
<router-view></router-view>
```

1. vue-router提供了一个全局组件 router-link
2. router-link实质上最终会渲染成a链接 to属性等价于提供 href属性(to无需#)
3. router-link提供了声明式导航高亮的功能(自带类名)

跳转传参

在跳转路由时, 可以给路由对应的组件内传值
在router-link上的to属性传值, 语法格式如下

- /path?参数名=值
- /path/值 – 需要路由对象提前配置 path: “/path/参数名”

对应页面组件接收传递过来的值

- $route.query.参数名
- $route.params.参数名

路由 - 重定向

> 匹配path后, 强制切换到目标path上

- 网页打开url默认hash值是/路径
- redirect是设置要重定向到哪个路由路径

例如: 网页默认打开, 匹配路由"/", 强制切换到"/find"上

```
const routes = [
  {
    path: "/", // 默认hash值路径
    redirect: "/find" // 重定向到/find
    // 浏览器url中#后的路径被改变成/find-重新匹配数组规则
  }
]
```

总结: 强制重定向后, 还会重新来数组里匹配一次规则

路由 - 404页面

路由最后, path匹配*(任意路径) – 前面不匹配就命中最后这个, 显示对应组件页面

```
import NotFound from '@/views/NotFound'
```

```
const routes = [
  // ...省略了其他配置
  // 404在最后(规则是从前往后逐个比较path)
  {
    path: "*",
    component: NotFound
  }
]
```

编程式导航

```
this.$router.push({
    path: "路由路径", // 都去 router/index.js定义
    name: "路由名"
})

this.$router.push({
    path: "路由路径"
    name: "路由名",
    query: {
            "参数名": 值
    }
    params: {
                "参数名": 值
    }
})

// 对应路由接收    $route.params.参数名    取值
// 对应路由接收    $route.query.参数名     取值
```

全局前置守卫

路由跳转之前,先执行一次前置守卫函数,判断是否可以正常跳转

```
// 目标: 路由守卫
// 场景: 当你要对路由权限判断时
```

```
// 语法: router.beforeEach((to, from, next)=>{//路由跳转"之前"先执行这里, 决定是否跳转})
// 参数1: 要跳转到的路由 (路由对象信息)     目标
// 参数2: 从哪里跳转的路由 (路由对象信息)   来源
// 参数3: 函数体 - next()才会让路由正常的跳转切换, next(false)在原地停留, next("强制修改到另一个路由路径上")
// 注意: 如果不调用next, 页面留在原地

// 例子: 判断用户是否登录, 是否决定去"我的音乐"/my
const isLogin = true; // 登录状态(未登录)
router.beforeEach((to, from, next) => {
  if (to.path === "/my" && isLogin === false) {
    alert("请登录")
    next(false) // 阻止路由跳转
  } else {
    next() // 正常放行
  }
})
```

全局解析守卫

你可以用 `router.beforeResolve` 注册一个全局守卫。这和 `router.beforeEach` 类似，因为它在 **每次导航**时都会触发，但是确保在导航被确认之前，**同时在所有组件内守卫和异步路由组件被解析之后，解析守卫就被正确调用。**这里有一个例子，确保用户可以访问自定义 meta 属性 `requiresCamera` 的路由：

```
router.beforeResolve(async to => {
  if (to.meta.requiresCamera) {
    try {
      await askForCameraPermission()
    } catch (error) {
      if (error instanceof NotAllowedError) {
        // ... 处理错误，然后取消导航
        return false
      } else {
```

```
        // 意料之外的错误，取消导航并把错误传给全局处理器
        throw error
      }
    }
  }
})
```

`router.beforeResolve` 是获取数据或执行任何其他操作（如果用户无法进入页面时你希望避免执行的操作）的理想位置。

全局后置钩子

你也可以注册全局后置钩子，然而和守卫不同的是，这些钩子不会接受 `next` 函数也不会改变导航本身：

```
router.afterEach((to, from) => {
  sendToAnalytics(to.fullPath)
})
```

它们对于分析、更改页面标题、声明页面等辅助功能以及许多其他事情都很有用。

它们也反映了 navigation failures 作为第三个参数：

```
router.afterEach((to, from, failure) => {
  if (!failure) sendToAnalytics(to.fullPath)
})
```

了解更多关于 navigation failures 的信息在它的指南中。

vuex

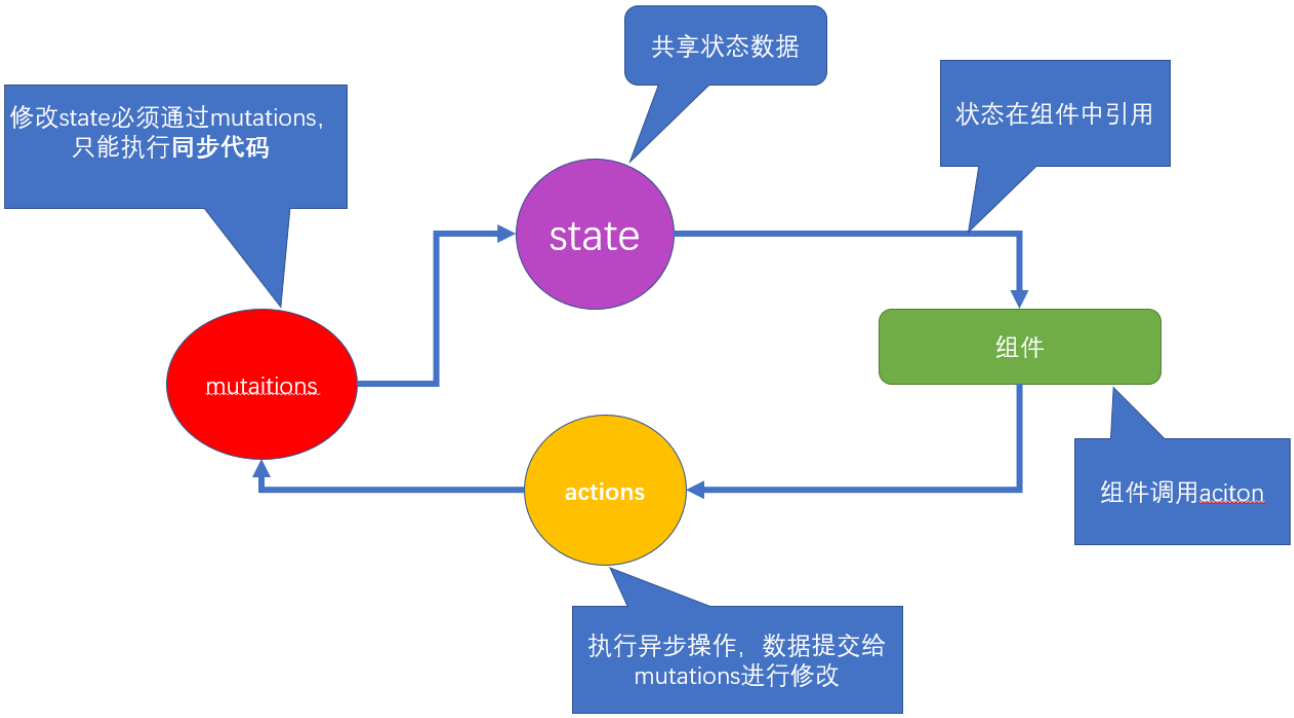
基础的组件通信的方式：

1. 父->子 `props`
2. 子->父 `$emit`
3. 兄弟之间 `eventBus` (使用的公共的$emit)

遇到其他的关系就不好解决！！

Vuex 是一个专为 Vue.js 应用程序开发的**状态管理模式。**它采用`集中式`存储管理应用的所有组件的状态，并以相应的规则保证状态以一种`可预测`的方式发生变化。

- vuex是采用集中式管理组件依赖的共享数据的一个工具，可以解决不同组件数据共享问题。



结论

1. 修改state状态必须通过`mutations`
2. `mutations`只能执行同步代码，类似ajax，定时器之类的代码不能在mutations中执行
3. 执行异步代码，要通过actions，然后将数据提交给mutations才可以完成
4. state的状态即共享数据可以在组件中引用
5. 组件中可以调用action

vuex基础-state

state是放置所有公共状态的属性，如果你有一个公共状态数据， 你只需要定义在 state对象中

定义state

```
// 初始化vuex对象
const store = new Vuex.Store({
  state: {
    // 管理数据
    count: 0
  }
})
```

辅助函数 - mapState

> mapState是辅助函数，帮助我们把store中的数据映射到组件的计算属性中, 它属于一种方便用法

vuex基础-mutations

state数据的修改只能通过mutations，并且mutations必须是同步更新，目的是形成`数据快照`
注意： Vuex中mutations中要求不能写异步代码，如果有异步的ajax请求，应该放置在actions中

vuex基础-actions

state是存放数据的，mutations是同步更新数据，actions则负责进行异步操作

vuex基础-getters

除了state之外，有时我们还需要从state中派生出一些状态，这些状态是依赖state的，此时会用到getters

Vuex中的模块化-Module

如果把所有的状态都放在state中，当项目变得越来越大的时候，Vuex会变得越来越难以维护

模块化中的命名空间

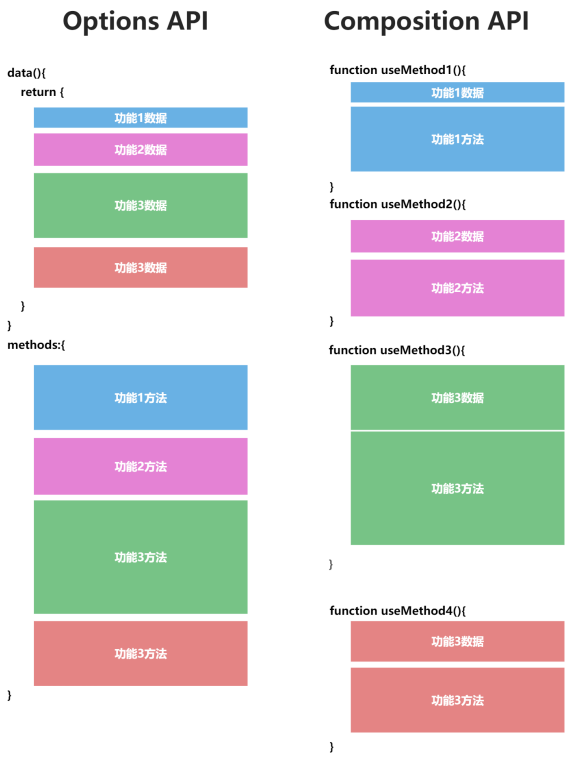
命名空间 `namespaced`
默认情况下，模块内部的 action、 mutation 和 getter 是注册在**全局命名空间**的——这样使得多个模块能够对同一 mutation 或 action 作出响应。

vuex中mapActions的使用

如果一个方法或多个方法需要在多个页面和组件中使用，那么，可以使用mapActions。

一般情况下我们会在组件中使用 this.$store.dispatch() 来触发 action，想要调用多少个 action 就需要调用多少次 dispatch()，而使用 mapActions 的话只需要往 mapActions 中传入与 action 同名的函数，然后调用这些函数即可触发对应的 action。

vue3



极客时间

选项API和组合API

什么是选项API写法：Options Apl

咱们在vue2.x项目中使用的就是 选项API 写法
代码风格：data选项写数据，methods选项写函数...，一个功能逻辑的代码分散。
优点：易于学习和使用，写代码的位置已经约定好
缺点：代码组织性差，相似的逻辑代码不便于复用，逻辑复杂代码多了不好阅读。

补充：虽然提供mixins用来封装逻辑，但是出现数据函数覆盖的概率很大，不好维护。

```
<template>
  <div class="container">
    <div>鼠标位置：</div>
    <div>X轴：{{x}}</div>
    <div>Y轴：{{y}}</div>
    <hr>
    <div>{{count}} <button @click="add()">自增
</button></div>
  </div>
</template>
<script>
export default {
  name: 'App', // data选项写数据
  data () {
    return {
      x: 0,
      y: 0,
      count: 0
    }
  },
  mounted() { // mounted生命周期钩子函数
    document.addEventListener('mousemove', this.move)
  },
  methods: { // methods选项写函数
    move(e) {
      this.x = e.pageX
      this.y = e.pageY
    },
    add () {
        this.count++
    }
  },
  destroyed() {
```

```
    document.removeEventListener('mousemove',
this.move)
  },
  compute:{
    // 计算属性是在computed
  }
}
</script>
```

什么是组合API写法：Compositon API
CompositionAPI的好处：

1. 所有 API 都是 import 引入的。用到的功能都 import 进来，对 Tree-shaking 很友好，代码中没用到功能，打包的时候会被清理掉，减小包的大小。
2. 不再上下反复横跳，我们可以把一个功能模块的 methods、data 都放在一起书写，维护更轻松。
3. 代码方便复用，可以把一个功能所有的 methods、data 封装在一个独立的**函数(不用为了一个小功能创建一个组件了**)里，复用代码非常容易。
   缺点：需要有良好的代码组织能力和拆分逻辑能力。
   补充：为了能让大家较好的过渡到vue3.0的版本来，也支持vue2.x选项API写法

```
<template>
  <div class="container">
    <div>鼠标位置：</div>
    <div>X轴：{{x}}</div>
    <div>Y轴：{{y}}</div>
    <hr>
    <div>{{count}} <button @click="add()">自增
</button></div>
  </div>
</template>
<script>
import { onMounted, onUnmounted, reactive, ref, toRefs
} from 'vue'
export default {
```

```
  name: 'App',
  setup () {
    // 鼠标移动逻辑
    const mouse = reactive({
      x: 0,
      y: 0
    })
    const move = e => {
      mouse.x = e.pageX
      mouse.y = e.pageY
    }
    onMounted(()=>{
      document.addEventListener('mousemove',move)
    })
    onUnmounted(()=>{
      document.removeEventListener('mousemove',move)
    })

    // 累加逻辑
    const count = ref(0)
    const add = () => {
      count.value ++
    }

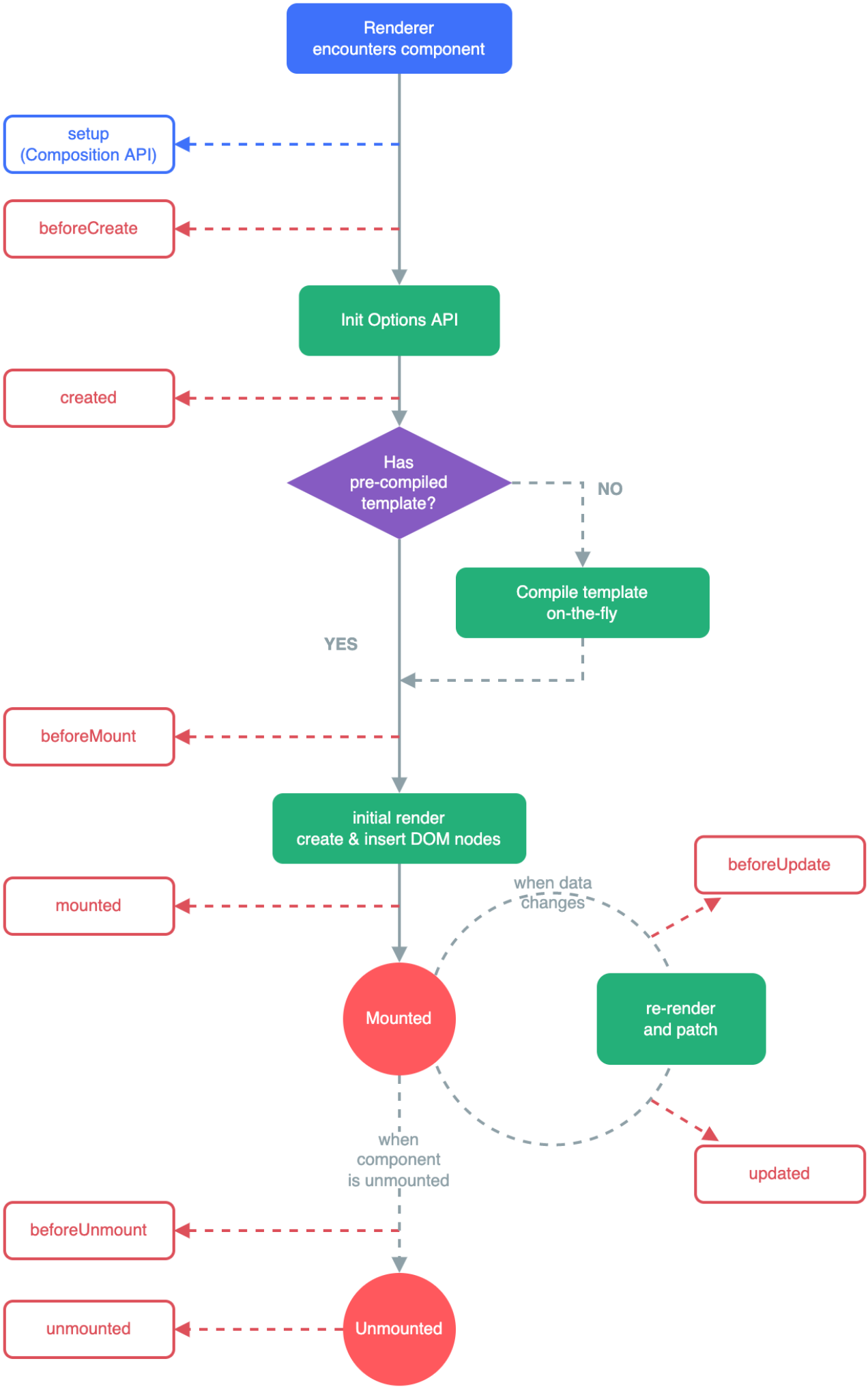
    // 返回数据
    return {
      ...toRefs(mouse),
      count,
      add
    }
  }
}
</script>
```

vue3.0的生命周期

回顾vue2.x生命周期钩子函数：

beforeCreate
created
beforeMount
mounted
beforeUpdate
updated
beforeDestroy
destroyed

认识vue3.0生命周期钩子函数


Renderer
encounters component
setup
(Composition API)
beforeCreate
Init Options API
created
Has
pre-compiled
template?
NO
Compile template
on-the-fly
YES
beforeMount
initial render
create & insert DOM nodes
beforeUpdate
when data
changes
mounted
Mounted
re-render
and patch
updated
when
component
is unmounted
beforeUnmount
unmounted
Unmounted

setup 创建实例前
onBeforeMount 挂载DOM前
onMounted 挂载DOM后
onBeforeUpdate 更新组件前
onUpdated 更新组件后
onBeforeUnmount 卸载销毁前
onUnmounted 卸载销毁后
总结： 组合API的生命周期钩子有7个，可以多次使用同一个钩子，执行顺序和书写顺序相同。
https://cn.vuejs.org/api/application.html

响应式对象

vue-响应式
Vue中对象的响应式原理解析
JS中有两种方法可以侦听变化：

Object.defineProperty —— Vue 2.x使用
https://v2.cn.vuejs.org/v2/guide/reactivity.html
Proxy —— Vue 3.x使用
https://cn.vuejs.org/guide/extras/reactivity-in-depth.html#what-is-reactivity

响应式总结

1. 所谓的响应式其实就是拦截对象属性的访问和设置，插入一些我们自己想要做的事情
2. 在Javascript中能实现响应式拦截的方法有俩种，`Object.defineProperty`方法和`Proxy对象代理`
3. 回归到vue2.x中的data配置项，只要放到了data里的数据，不管层级多深不管你最终会不会用到这个数据都会进行递归响应式处理，所以要求我们如非必要，尽量不要添加太多的冗余数据在data中
4. 需要了解vue3.x中，解决了2中对于数据响应式处理的无端性能消耗，使用的手段是Proxy劫持对象整体 + 惰性处理（用到了才进行响应式转换）

在CSS中使用JavaScript中的变量

我们可以通过 `v-bind` 函数来使用 JavaScript 中的变量去渲染样式，如果这个变量是响应式数据，就可以很方便地实现样式的切换。

```
<template>
  <div>
    <h1 @click="add">{{ count }}</h1>
  </div>
</template>

<script setup>
import { ref } from "vue";
let count = ref(1)
let color = ref('red')
function add() {
  count.value++
  color.value = Math.random()>0.5 ? "blue":"red"
}
</script>

<style scoped>
h1 {
  color:v-bind(color); /* 使用v-bind */
}
</style>>
```

reactive函数

- reactive是一个函数，它可以定义一个复杂数据类型，成为响应式数据。

```
<template>
  <div class="container">
    <div>{{obj.name}}</div>
    <div>{{obj.age}}</div>
    <button @click="updateName">修改数据</button>
  </div>
```

```
</template>
<script>
import { reactive } from 'vue'
export default {
  name: 'App',
  setup () {
    // 普通数据
    // const obj = {
    //   name: 'ls',
    //   age: 18
    // }
    const obj = reactive({
      name: 'ls',
      age: 18
    })

    // 修改名字
    const updateName = () => {
      console.log('updateName')
      obj.name = 'zs'
    }

    return { obj ,updateName}
  }
}
</script>
```

toRef函数

- toRef是函数，转换**响应式对象中某个**属性为单独响应式数据，并且**值是关联的。**

```
<template>
  <div class="container">
    {{name}} <button @click="updateName">修改数据
</button>
```

```
  </div>
</template>
<script>
import { reactive, toRef } from 'vue'
export default {
  name: 'App',
  setup () {
    // 1. 响应式数据对象
    const obj = reactive({
      name: 'ls',
      age: 10
    })
    console.log(obj)
    // 2. 模板中只需要使用name数据
    // 注意：从响应式数据对象中解构出的属性数据，不再是响应式数据
    // let { name } = obj 不能直接解构，出来的是一个普通数据
    const name = toRef(obj, 'name')
    // console.log(name)
    const updateName = () => {
      console.log('updateName')
      // toRef转换响应式数据包装成对象，value存放值的位置
      name.value = 'zs'
    }

    return {name, updateName}
  }
}
</script>
<style scoped lang="less"></style>
```

toRefs函数

- toRefs是函数，转换**响应式对象**中所有属性为单独响应式数据，对象成为普通对象，并且**值是关联的**

```vue
<template>
  <div class="container">
    <div>{{name}}</div>
    <div>{{age}}</div>
    <button @click="updateName">修改数据</button>
  </div>
</template>
<script>
import { reactive, toRef, toRefs } from 'vue'
export default {
  name: 'App',
  setup () {
    // 1. 响应式数据对象
    const obj = reactive({
      name: 'ls',
      age: 10
    })
    console.log(obj)
    // 2. 解构或者展开响应式数据对象
    // const {name,age} = obj
    // console.log(name,age)
    // const obj2 = {...obj}
    // console.log(obj2)
    // 以上方式导致数据就不是响应式数据了
    const obj3 = toRefs(obj)
    console.log(obj3)

    const updateName = () => {
      // obj3.name.value = 'zs'
      obj.name = 'zs'
    }

    return {...obj3, updateName}
  }
}
```

```
</script>
<style scoped lang="less"></style>
```

ref函数

定义响应式数据：

- ref函数，常用于简单数据类型定义为响应式数据
  - 再修改值，获取值的时候，需要.value
  - 在模板中使用ref申明的响应式数据，可以省略.value

```
<template>
  <div class="container">
    <div>{{name}}</div>
    <div>{{age}}</div>
    <button @click="updateName">修改数据</button>
  </div>
</template>
<script>
import { ref } from 'vue'
export default {
  name: 'App',
  setup () {
    // 1. name数据
    const name = ref('ls')
    console.log(name)
    const updateName = () => {
      name.value = 'zs'
    }
    // 2. age数据
    const age = ref(10)

    // ref常用定义简单数据类型的响应式数据
    // 其实也可以定义复杂数据类型的响应式数据
    // 对于数据未之的情况下 ref 是最适用的
    // const data = ref(null)
```

```
    // setTimeout(()=>{
    //   data.value = res.data
    // },1000)

    return {name, age, updateName}
  }
}
</script>
```