【讲义】Vue.js高级用法

#2021#

动画特效

1. transition 实现路由切换动画

- App.vue transition
- router/index.ts meta.depth
- Home.vue - HelloWorld.vue router-link
- List.vue
- Detail.vue
- home ➜ list ➜ detail
- detail ➜ list ➜ home

插槽 - Slot

Slot 插槽，我们可以理解为solt在组件模板中占好了位置，当使用该组件标签时候，组件标签里面的内容就会替换组件模板中slot位置，作为承载分发内容的出口

通过插槽可以让用户可以拓展组件，去更好地复用组件和对其做定制化处理

如果父组件在使用到一个复用组件的时候，获取这个组件在不同的地方有少量的更改，如果去重写组件是一件不明智的事情

通过slot插槽向组件内部指定位置传递内容，完成这个复用组件在不同场景的应用

比如布局组件、表格列、下拉选、弹框显示内容等

1. slot 使用

- 默认slot使用

子组件用标签来确定渲染的位置，标签里面可以放DOM结构，当父组件使用的时候没有往插槽传入内容，标签内DOM结构就会显示在页面

- 具名slot使用

子组件用v-slot来表示插槽的名字，不传为默认插槽

父组件中在使用时在默认插槽的基础上加上slot属性，值为子组件插槽v-slot的属性值

- 作用域slot使用

  组件间传递数据

- 一个基础的基于slot的页面布局
  - Slot-Container.vue
  - Page-1.vue
  - Page-2.vue
  - router/index.ts

2. slot 实现原理

   slot本质上是返回VNode的函数，一般情况下，Vue中的组件要渲染到页面上需要经过 template ➜ render function ➜ VNode ➜ DOM 过程。

   比如一个带slot的组件

```
Vue.component('button-counter', {
  template: '<div> <slot>我是默认内容</slot></div>'
})

new Vue({
    el: '#app',
    template: '<button-counter><span>我是slot传入内容</span></button-counter>',
    components:{buttonCounter}
})
```

   经过vue编译, 组件渲染函数会变成这样

```
(function anonymous(
) {
with(this){return _c('div',[_t("default",[_v("我是默认内容")])],2)}
```

```
})
```

而这个_t就是slot渲染函数:

```
function renderSlot (
  name,
  fallback,
  props,
  bindObject
) {
  // 得到渲染插槽内容的函数
  var scopedSlotFn = this.$scopedSlots[name];
  var nodes;
  // 如果存在插槽渲染函数，则执行插槽渲染函数，生成nodes节点返回
  // 否则使用默认值
  nodes = scopedSlotFn(props) || fallback;
  return nodes;
}
```

而scopedSlots其实就是递归解析各个节点, 获取slot

```
function resolveSlots (
    children,
    context
  ) {
    if (!children || !children.length) {
      return {}
    }
    var slots = {};
    for (var i = 0, l = children.length; i < l; i++) {
      var child = children[i];
      var data = child.data;
      // remove slot attribute if the node is resolved as a Vue slot node
      if (data && data.attrs && data.attrs.slot) {
        delete data.attrs.slot;
      }
      // named slots should only be respected if the vnode was rendered in the
```

```
      // same context.
      if ((child.context === context || child.fnContext === context) &&
        data && data.slot != null
      ) {
        // 如果slot存在(slot="header") 则拿对应的值作为key
        var name = data.slot;
        var slot = (slots[name] || (slots[name] = []));
        // 如果是tempalte元素 则把template的children添加进数组中，这也就是为什么你写的
template标签并不会渲染成另一个标签到页面
        if (child.tag === 'template') {
          slot.push.apply(slot, child.children || []);
        } else {
          slot.push(child);
        }
      } else {
        // 如果没有就默认是default
        (slots.default || (slots.default = [])).push(child);
      }
    }
    // ignore slots that contains only whitespace
    for (var name$1 in slots) {
      if (slots[name$1].every(isWhitespace)) {
        delete slots[name$1];
      }
    }
    return slots
}
```

Mixin

本质其实就是一个js对象，它可以包含我们组件中任意功能选项，如data、components、methods、created、computed等等

我们只要将共用的功能以对象的方式传入 mixins选项中，当组件使用 mixins对象时所有mixins对象的选项都将被混入该组件本身的选项中来

在Vue中我们可以 局部混入 和 全局混入, 全局混入常用于编写插件, 这个后面再说。

Tips:

- 当组件存在与mixin对象相同的数据的时候，进行递归合并的时候组件的数据会覆盖mixin的数据
- 如果相同数据为生命周期钩子的时候，会合并成一个数组，先执行mixin的钩子，再执行组件的钩子

1. mixin 使用

- mixins/index.ts 记录浏览页面时间
- Page-1.vue
- Page-2.vue

2. mixin 实现原理

- 优先递归处理 mixins
- 先遍历合并parent 中的key，调用mergeField方法进行合并，然后保存在变量options
- 再遍历 child，合并补上 parent 中没有的key，调用mergeField方法进行合并，保存在变量options
- 通过 mergeField 函数进行了合并

```
export function mergeOptions (
  parent: Object,
  child: Object,
  vm?: Component
): Object {

if (child.mixins) { // 判断有没有mixin 也就是mixin里面挂mixin的情况 有的话递归进行合并
    for (let i = 0, l = child.mixins.length; i < l; i++) {
    parent = mergeOptions(parent, child.mixins[i], vm)
    }
}

  const options = {}
  let key
  for (key in parent) {
    mergeField(key) // 先遍历parent的key 调对应的strats[XXX]方法进行合并
  }
```

```
  for (key in child) {
    if (!hasOwn(parent, key)) { // 如果parent已经处理过某个key 就不处理了
      mergeField(key) // 处理child中的key 也就parent中没有处理过的key
    }
  }
  function mergeField (key) {
    const strat = strats[key] || defaultStrat
    options[key] = strat(parent[key], child[key], vm, key) // 根据不同类型的
options调用strats中不同的方法进行合并
  }
  return options
}
```

其实主要的逻辑就是合并mixin和当前组件的各种数据, 细分为四种策略:

- 替换型策略 - 同名的props、methods、inject、computed会被后来者代替

```
strats.props =
strats.methods =
strats.inject =
strats.computed = function (
  parentVal: ?Object,
  childVal: ?Object,
  vm?: Component,
  key: string
): ?Object {
  if (!parentVal) return childVal // 如果parentVal没有值，直接返回childVal
  const ret = Object.create(null) // 创建一个第三方对象 ret
  extend(ret, parentVal) // extend方法实际是把parentVal的属性复制到ret中
  if (childVal) extend(ret, childVal) // 把childVal的属性复制到ret中
  return ret
}
```

- 合并型策略 - data, 通过set方法进行合并和重新赋值

```
```

```
strats.data = function(parentVal, childVal, vm) {
    return mergeDataOrFn(
        parentVal, childVal, vm
    )
};

function mergeDataOrFn(parentVal, childVal, vm) {
    return function mergedInstanceDataFn() {
        var childData = childVal.call(vm, vm) // 执行data挂的函数得到对象
        var parentData = parentVal.call(vm, vm)
        if (childData) {
            return mergeData(childData, parentData) // 将2个对象进行合并
        } else {
            return parentData // 如果没有childData 直接返回parentData
        }
    }
}

function mergeData(to, from) {
    if (!from) return to
    var key, toVal, fromVal;
    var keys = Object.keys(from);
    for (var i = 0; i < keys.length; i++) {
        key = keys[i];
        toVal = to[key];
        fromVal = from[key];
        // 如果不存在这个属性，就重新设置
        if (!to.hasOwnProperty(key)) {
            set(to, key, fromVal);
        }
        // 存在相同属性，合并对象
        else if (typeof toVal =="object" && typeof fromVal =="object") {
            mergeData(toVal, fromVal);
        }
    }
    return to
}
```

- 队列型策略 - 生命周期函数和watch，原理是将函数存入一个数组，然后正序遍历依次执行

```
function mergeHook (
  parentVal: ?Array<Function>,
  childVal: ?Function | ?Array<Function>
): ?Array<Function> {
  return childVal
    ? parentVal
      ? parentVal.concat(childVal)
      : Array.isArray(childVal)
        ? childVal
        : [childVal]
    : parentVal
}

LIFECYCLE_HOOKS.forEach(hook => {
  strats[hook] = mergeHook
})
```

- 叠加型策略 - component、directives、filters，通过原型链进行层层的叠加

```
strats.components=
strats.directives=

strats.filters = function mergeAssets(
    parentVal, childVal, vm, key
) {
    var res = Object.create(parentVal || null);
    if (childVal) {
        for (var key in childVal) {
            res[key] = childVal[key];
        }
    }
    return res
}
```

过滤器 - Filter

过滤器实质不改变原始数据，只是对数据进行加工处理后返回过滤后的数据再进行调用处理，我们也可以理解其为一个纯函数

Vue 允许你自定义过滤器，可被用于一些常见的文本格式化, 比如单位转换、数字打点、文本格式化、时间格式化等等。

但是Vue3中被弃用了, 建议使用computed实现这些功能.

1. filter 使用

- filters/index.ts 隐藏手机号中间4位
- Page-1.vue
- main.ts - 全局路由器
- 过滤器传参 & 多个过滤器串联使用

- 局部过滤器优先于全局过滤器被调用
- 一个表达式可以使用多个过滤器。过滤器之间需要用管道符“|”隔开。其执行顺序从左往右

2. filter 实现原理

- 在编译阶段通过parseFilters将过滤器编译成函数调用（串联过滤器则是一个嵌套的函数调用，前一个过滤器执行的结果是后一个过滤器函数的参数）

```
function parseFilters (filter) {
    let filters = filter.split('|')
    let expression = filters.shift().trim() // shift()删除数组第一个元素并将其返回，该方法会更改原数组
    let i
    if (filters) {
        for(i = 0;i < filters.length;i++){
            experssion = warpFilter(expression,filters[i].trim()) // 这里传进去的expression实际上是管道符号前面的字符串，即过滤器的第一个参数
        }
    }
    return expression
```

```
}
// warpFilter函数实现
function warpFilter(exp,filter){
    // 首先判断过滤器是否有其他参数
    const i = filter.indexof('(')
    if(i<0){ // 不含其他参数，直接进行过滤器表达式字符串的拼接
        return `_f("${filter}")(${exp})`
    }else{
        const name = filter.slice(0,i) // 过滤器名称
        const args = filter.slice(i+1) // 参数，但还多了 ‘)’
        return `_f('${name}')(${exp},${args}` // 注意这一步少给了一个 ')'
    }
}
```

- 编译后通过调用resolveFilter函数找到对应过滤器并返回结果

```
export function resolveFilter(id){
    return resolveAsset(this.$options,'filters',id,true) || identity
}
export function resolveAsset(options,type,id,warnMissing){ // 因为我们找的是过滤器，所以在 resolveFilter函数中调用时 type 的值直接给的 'filters',实际这个函数还可以拿到其他很多东西
    if(typeof id !== 'string'){ // 判断传递的过滤器id 是不是字符串，不是则直接返回
        return
    }
    const assets = options[type]  // 将我们注册的所有过滤器保存在变量中
    // 接下来的逻辑便是判断id是否在assets中存在，即进行匹配
    if(hasOwn(assets,id)) return assets[id] // 如找到，直接返回过滤器
    // 没有找到，代码继续执行
    const camelizedId  = camelize(id) // 万一你是驼峰的呢
    if(hasOwn(assets,camelizedId)) return assets[camelizedId]
    // 没找到，继续执行
    const PascalCaseId = capitalize(camelizedId) // 万一你是首字母大写的驼峰呢
    if(hasOwn(assets,PascalCaseId)) return assets[PascalCaseId]
    // 如果还是没找到，则检查原型链(即访问属性)
    const result = assets[id] || assets[camelizedId] || assets[PascalCaseId]
    // 如果依然没找到，则在非生产环境的控制台打印警告
```

```
    if(process.env.NODE_ENV !== 'production' && warnMissing && !result){
        warn('Failed to resolve ' + type.slice(0,-1) + ': ' + id, options)
    }
    // 无论是否找到，都返回查找结果
    return result
}
```

- 执行结果作为参数传递给toString函数，而toString执行后，其结果会保存在Vnode的text属性中，渲染到视图

```
function toString(value){
    return value == null
    ? ''
    : typeof value === 'object'
      ? JSON.stringify(value,null,2)// JSON.stringify()第三个参数可用来控制字符串里面的间距
      : String(value)
}
```

插件 - Plugin

简单来说，插件就是指对Vue的功能的增强或补充。

1. 使用剪贴板插件

- mixins/index.ts
- Page-1.vue
- 查看剪贴板插件的源码

2. 什么是插件? 如何编写一个插件?

```
MyPlugin.install = function (Vue, options) {
  // 1. 添加全局方法或 property
  Vue.myGlobalMethod = function () {
    // 逻辑...
```

```
  }

  // 2. 添加全局资源
  Vue.directive('my-directive', {
    bind (el, binding, vnode, oldVnode) {
      // 逻辑...
    }
    ...
  })

  // 3. 注入组件选项
  Vue.mixin({
    created: function () {
      // 逻辑...
    }
    ...
  })

  // 4. 添加实例方法
  Vue.prototype.$myMethod = function (methodOptions) {
    // 逻辑...
  }
}


Vue.use(plugin, options);
```

3. Vue.use做了什么?

- 判断当前插件是否已经安装过, 防止重复安装
- 处理参数, 调用插件的install方法, 第一个参数是Vue实例.

```
// Vue源码文件路径: src/core/global-api/use.js

import { toArray } from '../util/index'

export function initUse (Vue: GlobalAPI) {
```

```
  Vue.use = function (plugin: Function | Object) {
    const installedPlugins = (this._installedPlugins || (this._installedPlugins
= []))
    if (installedPlugins.indexOf(plugin) > -1) {
      return this
    }

    // additional parameters
    const args = toArray(arguments, 1)
    args.unshift(this)
    if (typeof plugin.install === 'function') {
      plugin.install.apply(plugin, args)
    } else if (typeof plugin === 'function') {
      plugin.apply(null, args)
    }
    installedPlugins.push(plugin)
    return this
  }
}
```

常见组件库介绍

- vant https://vant-contrib.gitee.io/vant/#/zh-CN/
- iview https://iview.github.io/
- element-ui https://element.eleme.cn/#/zh-CN/component/installation
- ant-design-vue https://www.antdv.com/docs/vue/introduce-cn/
- vue-material https://www.creative-tim.com/vuematerial/