11.27 【课件】Vue3 入门

本节课的主题

对比 vue2，对 vue3 的新特性进行学习，掌握 vue3 的关键技术点，以期能够使用 vue3 实现组件的开发。

核心概念

我们知道 Vue2 是响应式原理基于 Object.defineProperty 方法重定义对象的 getter 与 setter，vue3 则基于 Proxy 代理对象，拦截对象属性的访问与赋值过程。差异在于，前者并不能对诸如数组长度变化、增删元素操作、对象新增属性进行感知，在 vue 层面不得不重写一些数组方法（push、pop、unshift、shift 等），动态添加响应式属性，也要使用 $set 方法等。而 Proxy 则完美地从根上解决了这些问题，不过对于不支持 Proxy 对象的浏览器（如 IE），如果要使用 vue3 依然要进行降级兼容。

1. Object.defineProperty

```
// 假设我们在 data 函数中返回的数据为 initData
const initData = { value: 1 };

// 基于 initData 创建响应式对象 data
const data = {};

Object.keys(initData).forEach(key => {
  Object.defineProperty(data, key, {
    get() {
      // 此处依赖收集
       console.log('访问了', key);
       return initData[key];
    },
    set(v) {
       // 此处执行回调更新
       console.log('访问了', key);
       initData[key] = v;
    }
  });
});

// 从这里可以看出，initData 动态添加的属性，并不能被观测到，这也是 Vue.$set 存在的原因。
```

2. Proxy

```
const initData = { value: 1 };

const proxy = new Proxy(initData, {
  get(target, key) {
    // 此处依赖收集
     console.log('访问了', key);
     return target[key];
  },
  set(target, key, value) {
    // 此处执行回调更新
    console.log('修改了', key);
    return Reflect.set(target, key, value);
  }
});
// Proxy 可以观测到动态添加的属性
```

安装

```
$ npm init vite-app <project-name>
$ cd <project-name>
$ npm install
$ npm run dev
```

或者

```
$ yarn create vite-app <project-name>
$ cd <project-name>
$ yarn
$ yarn dev
```

vue3 新特性

异步组件需要使用 `defineAsyncComponent` 创建

全局注册

```
const AsyncComp = defineAsyncComponent(() =>
      import('./components/AsyncComp.vue')
)
app.component('async-component', AsyncComp)
```

局部声明

```
const AsyncComp = defineAsyncComponent(() =>
      import('./components/AsyncComp.vue')
)
...
{
  components:  { 'async-comp': AsyncComp }
}
```

自定义指令更新

```
自定义指令的钩子
// vue2
bind - 指令绑定到元素后发生。只发生一次。
inserted - 元素插入父 DOM 后发生。
update - 当元素更新，但子元素尚未更新时，将调用此钩子。
componentUpdated - 一旦组件和子级被更新，就会调用这个钩子。
unbind - 一旦指令被移除，就会调用这个钩子。也只调用一次。.

// vue3
bind → beforeMount
inserted → mounted
beforeUpdate：新的！这是在元素本身更新之前调用的，很像组件生命周期钩子。
update → 移除！有太多的相似之处要更新，所以这是多余的，请改用 updated。
componentUpdated → updated
beforeUnmount：新的！与组件生命周期钩子类似，它将在卸载元素之前调用。
unbind -> unmounted

上面的差异在 vue3 的官方文档上已经指出，课上主要对这些差异进行讲解和学习。
```

如果我们要实现一个指令，使用它的输入框挂载后立即聚焦，移除时发出信息（假设指令为 v-focus）：

```
// 用例：

<input v-focus />

// before:

  Vue.directive('focus', {
    bind(el, binding, vnode) {
      console.log(vnode.context)
```

```
    },
    inserted(el) {
      el.focus()
    },
    unbind() {
      alert('我被卸载了')
    }
  })

// after
import { createApp } from 'vue'

const app = createApp({})

app.directive('focus', {
    beforeMount(el, binding, vnode, prevVnode) {
      console.log(binding.instance)
    },
    mounted(el) {
      el.focus()
    },
    unmounted() {
      alert('我被卸载了')
    }
})
```

Teleport

将组件的 DOM 结构“传送”到指定的节点，脱离组件的父子关系，例如弹框的渲染。

```
// 代码结构
<div class="app">
    <teleport to="body">
      <input v-focus v-if="destroy" />
    </teleport>
</div>
// 真实的 HTML 结构
```

Data

data 声明不再接收纯 JavaScript object，而必须使用 function 声明返回。

```
export default {
  data() {
    return {
      state: '',
    }
  }
}
```

多根节点组件、函数式组件

多根节点允许 template 标签内直接出现多个子集标签，注意默认根节点需要明确指定。

```
<template>
  <header>...</header>
  <main v-bind="$attrs">...</main>
  <footer>...</footer>
</template>
// $attrs 默认本来绑定在唯一根节点上，多节点时需要明确指定挂在哪个节点下
```

在 Vue 2 中，函数式组件有两个主要用例：

1. 作为性能优化，因为它们的初始化速度比有状态组件快得多
2. 返回多个根节点

Vue 3 中，函数式组件剩下的唯一用例就是简单组件。只能使用普通函数来声明。

```
// vue2  Functional.vue
<template>
  <div>{{ msg }} </div>
</template>
<script>
```

```
  export default {
    functional: true,
    props: ['msg']
  }
</script>

// vue3  Functional.js

import { h } from 'vue'

export default function Functional(props, context) {
  return h('div',
    context.attrs,
    props.msg
  )
}
Functional.props = ['msg']
```

一些其他与开发者关系较为密切的更新

在同一元素上使用的 v-if 和 v-for 优先级已更改。即 if 比 for 的优先级更高了；

<template v-for> 和非 v-for 节点上 key 用法已更改；

渲染函数不再提供 createElement 参数（通常简写为 h），而是从依赖中导入；

标签上的属性与绑定对象的属性冲突时，以排在后面的属性为准，不再给单独属性更高的优先级。例如：

```
  <your-comp name="Tom" v-bind="{ name: 'Jim' }" />

  // vue2 your-comp 的 props:
  name: 'Tom'

  // vue3 your-comp 的 props:
  name: 'Jim'

  如果顺序是 <your-comp v-bind="{ name: 'Jim' }" name="Tom" />
  则 vue2 和 vue3 的 props 均为
  name: 'Tom'
```

这一点也契合了对象扩展运算合并时后者覆盖前者的哲学思想，减少开发者的心智负担。更多的细微变化，可以结合官方文档全面了解。

最具有颠覆意义的响应-组合 API

这也是我们学习和转向 vue3 最有意义的一部分内容，可以这么说，如果没有掌握这一部分内容，那么即使我们底层使用了 vue3，也不过是写 vue2。就像我们拥有了一台铲车，却还在坚持靠人力把重物装卸到车叉上，只把铲车当运输工具一样。

我们先来认识这几个新的 api（蓝色为重点内容，必须掌握）：

setup 函数

组件创建前执行的初始化函数，默认参数包含 props 和 context。context 可以理解为组件实例挂载之前的上下文（虽然这么理解，但不是实例本身，这点要清楚），所以实例上的 `attrs`，`slots`，`emit` 可以在 context 上访问到。

reactive | readonly | shallowReactive | shallowReadonly

用于创建响应式的对象，isReactive 可以判断对象是否为响应式的。isProxy 区分是哪种方法创建的代理对象，isReadonly 判断对象是否为 **readonly** 创建。shallowXXX 根据名称可知，只将对象自身的浅层属性进行转换，深层属性保持不变。

toRaw | markRaw

前者返回代理对象的源对象，后者将普通对象转换为不可代理的对象。

ref | toRef | unref | toRefs | isRef | shallowRef | triggerRef

常用于包装基本类型值为响应式对象，例如 `const box = ref(0)`，`box.value === 0`

unref：`isRef(data) ? data.value : data`

shallowRef：创建浅层的包装对象

triggerRef：人为触发订阅了 **shallowRef** 的副作用。

effect | watch | watchEffect

对数据源的更新订阅，一旦订阅的数据发生变化，将自动执行副作用函数。效果与 vue2 的 watch 配置类似。

computed

基于响应式数据生成衍生数据，且衍生数据会同步响应式数据的变化，效果与 vue2 的 computed 属性一样。

provide，inject

跨组件传递数据的方案。

基于 setup 方法使用的生命周期钩子同样有对应更新

~~beforeCreate~~ -> setup()

~~created~~ -> setup()

```
beforeMount -> onBeforeMount

mounted -> onMounted

beforeUpdate -> onBeforeUpdate

updated -> onUpdated

beforeUnmount -> onBeforeUnmount

unmounted -> onUnmounted

errorCaptured -> onErrorCaptured // 错误上报钩子，仅做了解

renderTracked -> onRenderTracked // { key, target, type } 仅做了解

renderTriggered -> onRenderTriggered // { key, target, type } 仅做了解
```

这些更新的使用方法也比较简单，我们将会在后续的实践环节逐渐学习。

```
// Composition.vue

<template>
  <div class="Composition">
    {{title}}
    <button @click="click">点击 ref: {{count}}</button>
    <button @click="increment">点击 reactive: {{number.num}}</button>
    <button @click="decrease">减少 reactive</button>
    <span>computed: {{total}}</span>
  </div>
</template>

<script>
import { ref, reactive, computed, effect, watch } from 'vue'

export default {
  props: ['title'],
  setup(props, context) {
    const count = ref(0)
    const number = reactive({ num: 1 })

    const click = e => {
      count.value++
```

```
    }
    const increment = e => {
      number.num++
    }

    const decrease = e => {
      number.num--
    }

    const total = computed(() => count.value + number.num)

    effect(() => console.log(count.value))

    watch(
      () => total.value > 5,
      exceeded => {
        console.log('第一个参数是函数，只有返回值变化了才会执行一次')
        if (exceeded) {
          alert('超过了')
        }
      }
    );

    watch(
      total,
      (current, prev) => {
        console.log(current, prev, '第一个参数不是函数，值变化一次就会执行一次')
      }
    );

    return {
      count, click,
      number, increment,
      total, decrease
    }
  }
}
</script>
```

强大的代码复用能力 hook

```
import { ref, onMounted, watch } from 'vue'

const fetch = (id) => new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve({ name: 'user-' + Math.random(), id});
  }, 2000)
})

export default function useUserInfo(id) { // id 为响应式数据源
  const info = ref({})
  const loading = ref(false)

  const getUserInfo = () => {
    loading.value = true
    fetch(id.value).then(user => {
      info.value = user
      loading.value = false
    })
  }

  onMounted(getUserInfo)

  watch(() => id.value, getUserInfo);

  return { info, loading }
}
```

跨组件传递数据

```
  // 常规组件中
  provide() {
    return {
      state: this.state,
      constant: '常量'
    }
  }
  // setup 方法中

  import { provide, readonly, inject } from 'vue'
  provide(key, reactiveValue)

  // 禁止子组件对 reactiveValue 进行修改的话

  provide(key, readonly(reactiveValue))
```

```
// 子组件使用数据
{
  inject: [key],
}
// 或使用组合 API
inject('state');
```

终极简化：单文件组件 <script setup>

defineProps，defineEmits 直接在<script setup>中使用，无须导入。

useSlots 和 useAttrs 是真实的运行时函数，它会返回与 setupContext.slots 和 setupContext.attrs 等价的值，同样也能在普通的组合式 API 中使用。

```
// 在<script setup>内部的顶层变量，均能直接用于模板部分，
// 省略了 setup 函数及其返回值

<template>
    <span @click="increment">{{count}}</span>
</template>
<script setup>
    import { ref } from 'vue'
    const count = ref(0)
    const increment = () => {
        count.value++
    }
</script>
```