基础

创建应用

1. 可以创建一个Vue实例，也可以创建多个Vue实例
2. `createApp` API **允许你在同一个页面中创建多个共存的 vue 应用，而且每个应用都拥有自己的用于配置和全局资源的作用域。**

```
<div id="app"></div>
<div id="app2"></div>
```

```
import { createApp } from 'vue'
// 从一个单文件组件中导入根组件
import App from './App.vue'
import HelloWorld from './components/HelloWorld.vue'

const app = createApp(App)
app.component('HelloWorld', HelloWorld).mount('#app')
const app2 = createApp(App)
app2.mount('#app2')
```

模板语法

若想插入 HTML，你需要使用 `v-html` 指令

```
<p>Using text interpolation: {{ rawHtml }}</p>
<p>Using v-html directive: <span v-html="rawHtml"></span></p>
```

 image-20240719152640307

在网站上动态渲染任意 HTML 是非常危险的，因为这非常容易造成 XSS 漏洞。请仅在内容安全可信时再使用 `v-html`，并且**永远不要**使用用户提供的 HTML 内容。

属性的动态绑定多个值

```
const objectOfAttrs = {
  id: 'container',
  class: 'wrapper',
  style: 'background-color:green'
}
<div v-bind="objectOfAttrs"></div>
```

{{}}仅支持表达式

app.config.globalProperties：一个用于注册能够被应用内所有组件实例访问到的全局属性的对象。

这是对 Vue 2 中 `Vue.prototype` 使用方式的一种替代，此写法在 Vue 3 已经不存在了。与任何全局的东西一样，应该谨慎使用。

这个api的使用

```
const app = createApp(App);
app.config.globalProperties.$user = {
    name: '梅长苏',
    weapons: '长剑',
    title: '刺客'
}
//模板使用
<p>姓名：{{$user.name}} </p>
//setup里面使用
const cns = getCurrentInstance()
console.log(cns.appContext.config.globalProperties.$user)

 // or
 const {proxy} = getCurrentInstance()
 console.log(proxy.$user)

//
globalProperties和provide的使用区别

globalProperties是挂载在vue实例上面的，所以可以直接在template里面访问
provide/inject 是为vue组件通讯设计的一对方法，需要显示的声明之后才能使用
```

动态参数：同样在指令参数上也可以使用一个 JavaScript 表达式，需要包含在一对方括号内

```
<!--
注意，参数表达式有一些约束，
参见下面“动态参数值的限制”与“动态参数语法的限制”章节的解释
-->
<a v-bind:[attributeName]="url"> ... </a>

<!-- 简写 -->
<a :[attributeName]="url"> ... </a>

这里的 attributeName 会作为一个 JavaScript 表达式被动态执行，计算得到的值会被用作最终的参数。举例来说，如果你的组件实例有一个数据属性 attributeName，其值为 "href"，那么这个绑定就等价于 v-bind:href。
```

类和样式绑定

```
<div
  class="static"
  :class="{ active: isActive, 'text-danger': hasError }"
></div>
```

通过对象的形式绑定

```
const classObject = reactive({
  active: true,
  'text-danger': false
})
<div :class="classObject"></div>
```

或者计算属性

```
const isActive = ref(true)
const error = ref(null)

const classObject = computed(() => ({
  active: isActive.value && !error.value,
  'text-danger': error.value && error.value.type === 'fatal'
}))
```

绑定数组

```
const activeClass = ref('active')
const errorClass = ref('text-danger')
<div :class="[activeClass, errorClass]"></div>

//三元表达式
<div :class="[isActive ? activeClass : '', errorClass]"></div>
//嵌套对象
<div :class="[{ [activeClass]: isActive }, errorClass]"></div>
```

渗透

```
<!-- MyComponent 模板使用 $attrs 时 -->
<p :class="$attrs.class">Hi!</p>
<span>This is a child component</span>
<!-- 父组件 -->
<MyComponent class="baz" />
```

样式

```
<div :style="{ 'font-size': fontSize + 'px' }"></div>
```

绑定对象

```
const styleObject = reactive({
  color: 'red',
  fontSize: '30px'
})
<div :style="styleObject"></div>
```

绑定数组

```
<div :style="[baseStyles, overridingStyles]"></div>
```

计算属性

1. 有缓存
2. **不要改变其他状态、在 getter 中做异步请求或者更改 DOM**!
3. getter 的职责应该仅为计算和返回该值

4. 计算属性的返回值应该被视为只读的，并且永远不应该被更改——应该更新它所依赖的源状态以触发新的计算。

条件渲染

`v-if` vs. `v-show`

`v-if` 是“真实的”按条件渲染，因为它确保了在切换时，条件区块内的事件监听器和子组件都会被销毁与重建。

`v-if` 也是**惰性**的：如果在初次渲染时条件值为 false，则不会做任何事。条件区块只有当条件首次变为 true 时才被渲染。

相比之下，`v-show` 简单许多，元素无论初始条件如何，始终会被渲染，只有 CSS `display` 属性会被切换。

总的来说，`v-if` 有更高的切换开销，而 `v-show` 有更高的初始渲染开销。因此，如果需要频繁切换，则使用 `v-show` 较好；如果在运行时绑定条件很少改变，则 `v-if` 会更合适。

v-if和v-show

当 `v-if` 和 `v-for` 同时存在于一个元素上的时候，`v-if` 会首先被执行。

列表渲染

v-for可以遍历对象

通过key管理状态

Vue 默认按照“就地更新”的策略来更新通过 `v-for` 渲染的元素列表。当数据项的顺序改变时，Vue 不会随之移动 DOM 元素的顺序，而是就地更新每个元素，确保它们在原本指定的索引位置上渲染。

默认模式是高效的，但**只适用于列表渲染输出的结果不依赖子组件状态或者临时 DOM 状态 (例如表单输入值) 的情况。**

为了给 Vue 一个提示，以便它可以跟踪每个节点的标识，从而重用和重新排序现有的元素，你需要为每个元素对应的块提供一个唯一的 `key` attribute。

在没有 key 的情况下，Vue 将使用一种最小化元素移动的算法，并尽可能地就地更新/复用相同类型的元素。如果传了 key，则将根据 key 的变化顺序来重新排列元素，并且将始终移除/销毁 key 已经不存在的元素。

同一个父元素下的子元素必须具有**唯一的 key**。重复的 key 将会导致渲染异常。

key：

- 触发特定的声明钩子
- 触发过渡

事件处理

修饰符

- 表单

lazy、trim、number

lazy:

在我们填完信息，光标离开标签的时候，才会将值赋予给`value`，也就是在`change`事件之后再进行信息同步

trim:

自动过滤用户输入的首空格字符，而中间的空格不会过滤

number:

自动将用户的输入值转为数值类型，但如果这个值无法被`parseFloat`解析，则会返回原来的值

- 事件
- stop
- prevent
- self
- once
- capture
- passive
- native

```
<!-- 滚动事件的默认行为 (即滚动行为) 将会立即触发 -->
<!-- 而不会等待 `onScroll` 完成  -->
<!-- 这其中包含 `event.preventDefault()` 的情况 -->
<div v-on:scroll.passive="onScroll">...</div>
```

让组件变成像`html`内置标签那样监听根元素的原生事件，否则组件上使用`v-on`只会监听自定义事件

```
<my-component v-on:click.native="doSomething"></my-component>
```

- 鼠标
- left、right、middle
- v-bind修饰符

  async

  能对`props`进行一个双向绑定

  ```
  //父组件
  <comp :myMessage.sync="bar"></comp>
  //子组件
  this.$emit('update:myMessage',params);
  ```

  相当于

```
//父亲组件
<comp :myMessage="bar" @update:myMessage="func"></comp>
func(e){
 this.bar = e;
}
//子组件js
func2(){
  this.$emit('update:myMessage',params);
}
```

watchEffect和watch区别

你可以参考一下[这个例子]的 `watchEffect` 和响应式的数据请求的操作。

对于这种只有一个依赖项的例子来说，`watchEffect()` 的好处相对较小。但是对于有多个依赖项的侦听器来说，使用 `watchEffect()` 可以消除手动维护依赖列表的负担。此外，如果你需要侦听一个嵌套数据结构中的几个属性，`watchEffect()` 可能会比深度侦听器更有效，因为它将只跟踪回调中被使用到的属性，而不是递归地跟踪所有的属性。

- `watch` 只追踪明确侦听的数据源。它不会追踪任何在回调中访问到的东西。另外，仅在数据源确实改变时才会触发回调。`watch` 会避免在发生副作用时追踪依赖，因此，我们能更加精确地控制回调函数的触发时机。
- `watchEffect`，则会在副作用发生期间追踪依赖。它会在同步执行过程中，自动追踪所有能访问到的响应式属性。这更方便，而且代码往往更简洁，但有时其响应性依赖关系会不那么明确。

post Watchers

如果想在侦听器回调中能访问被 Vue 更新**之后**的所属组件的 DOM，你需要指明 `flush: 'post'` 选项：

```
watch(source, callback, {
  flush: 'post'
})

watchEffect(callback, {
  flush: 'post'
})
```

后置刷新的 `watchEffect()` 有个更方便的别名 `watchPostEffect()`

同步侦听器

```
watch(source, callback, {
  flush: 'sync'
})

watchEffect(callback, {
  flush: 'sync'
})
```

同步触发的 `watchEffect()` 有个更方便的别名 `watchSyncEffect()`

模板的引用

函数模板引用

除了使用字符串值作名字，`ref` attribute 还可以绑定为一个函数，会在**每次组件更新时都被调用。**该函数会收到元素引用作为其第一个参数

```
<input :ref="(el) => { /* 将 el 赋值给一个数据属性或 ref 变量 */ }">
```

动态组件

有些场景会需要在两个组件间来回切换，比如 Tab 界面：

```
<!-- currentTab 改变时组件也改变 -->
<component :is="tabs[currentTab]"></component>
```

当使用 `<component :is="...">` 来在多个组件间作切换时，被切换掉的组件会被卸载。我们可以通过 `keep-alive` 组件强制被切换掉的组件仍然保持“存活”的状态。

深入组件

注册

1. 全局注册

```
app
  .component('ComponentA', ComponentA)
  .component('ComponentB', ComponentB)
  .component('ComponentC', ComponentC)
```

缺点:

1. 全局注册，但并没有被使用的组件无法在生产打包时被自动移除 (也叫“tree-shaking”)。如果你全局注册了一个组件，即使它并没有被实际使用，它仍然会出现在打包后的 JS 文件中。
2. 全局注册在大型项目中使项目的依赖关系变得不那么明确。在父组件中使用子组件时，不太容易定位子组件的实现。和使用过多的全局变量一样，这可能会影响应用长期的可维护性。
3. 局部注册

为了方便，Vue 支持将模板中使用 kebab-case 的标签解析为使用 PascalCase 注册的组件。这意味着一个以 `MyComponent` 为名注册的组件，在模板中可以通过 `<MyComponent>` 或 `<my-component>` 引用。

Props

1. 绑定多个props

```
const post = {
  id: 1,
  title: 'My Journey with Vue'
}
<BlogPost v-bind="post" />
    =>
<BlogPost :id="post.id" :title="post.title" />
```

2. 遵循单向数据流:所有的 props 都遵循着**单向绑定**原则，props 因父组件的更新而变化，自然地将新的状态向下流往子组件，而不会逆向传递。这避免了子组件意外修改父组件的状态的情况，不然应用的数据流将很容易变得混乱而难以理解。

   另外，每次父组件更新后，所有的子组件中的 props 都会被更新到最新值，这意味着你**不应该**在子组件中去更改一个 prop。若你这么做了，Vue 会在控制台上向你抛出警告

3. 更改props两种情况:

prop 被用于传入初始值；而子组件想在之后将其作为一个局部数据属性。在这种情况下，最好是新定义一个局部数据属性，从 props 上获取初始值即可：

```
const props = defineProps(['initialCounter'])

// 计数器只是将 props.initialCounter 作为初始值
// 像下面这样做就使 prop 和后续更新无关了
const counter = ref(props.initialCounter)
```

需要对传入的 prop 值做进一步的转换。在这种情况中，最好是基于该 prop 值定义一个计算属性：

```
const props = defineProps(['size'])

// 该 prop 变更时计算属性也会自动更新
const normalizedSize = computed(() => props.size.trim().toLowerCase())
```

4. 更改数组或者对象的props

当对象或数组作为 props 被传入时，虽然子组件无法更改 props 绑定，但仍然**可以**更改对象或数组内部的值。这是因为 JavaScript 的对象和数组是按引用传递，对 Vue 来说，阻止这种更改需要付出的代价异常昂贵。

这种更改的主要缺陷是它允许了子组件以某种不明显的方式影响父组件的状态，可能会使数据流在将来变得更难以理解

5. props校验

```
defineProps({
  // 基础类型检查
  // （给出 `null` 和 `undefined` 值则会跳过任何类型检查）
  propA: Number,
  // 多种可能的类型
  propB: [String, Number],
  // 必传，且为 String 类型
  propC: {
    type: String,
    required: true
  },
```

```
  // 必传但可为 null 的字符串
  propD: {
    type: [String, null],
    required: true
  },
  // Number 类型的默认值
  propE: {
    type: Number,
    default: 100
  },
  // 对象类型的默认值
  propF: {
    type: Object,
    // 对象或数组的默认值
    // 必须从一个工厂函数返回。
    // 该函数接收组件所接收到的原始 prop 作为参数。
    default(rawProps) {
      return { message: 'hello' }
    }
  },
  // 自定义类型校验函数
  // 在 3.4+ 中完整的 props 作为第二个参数传入
  propG: {
    validator(value, props) {
      // The value must match one of these strings
      return ['success', 'warning', 'danger'].includes(value)
    }
  },
  // 函数类型的默认值
  propH: {
    type: Function,
    // 不像对象或数组的默认，这不是一个
    // 工厂函数。这会是一个用来作为默认值的函数
    default() {
      return 'Default function'
    }
  }
})
```

> `defineProps()` 宏中的参数**不可以访问** `<script setup>` **中定义的其他变量**，因为在编译时整个表达式都会被移到外部的函数中。

```
defineProps({
  author: Person
})
```

Vue 会通过 `instanceof Person` 来校验 `author` prop 的值是否是 `Person` 类的一个实例。

事件

1. 模板里面可以通过`$emit`来调用
2. 标注

```
<script setup lang="ts">
const emit = defineEmits<{
  (e: 'change', id: number): void
  (e: 'update', value: string): void
}>()
</script>
```

> 如果一个原生事件的名字 (例如 `click`) 被定义在 `emits` 选项中，则监听器只会监听组件触发的 `click` 事件而不会再响应原生的 `click` 事件。

3. 事件校验

```
<script setup>
const emit = defineEmits({
  // 没有校验
  click: null,

  // 校验 submit 事件
  submit: ({ email, password }) => {
    if (email && password) {
      return true
    } else {
      console.warn('Invalid submit event payload!')
      return false
    }
  }
})

function submitForm(email, password) {
  emit('submit', { email, password })
}
</script>
```

组件v-modal(更新了defineModal)

1. 3.4之后推荐使用`defineModal`
2. 示例

```
<!--父组件 -->
<MyComponent v-model:title="bookTitle" />

<!-- MyComponent.vue -->
<script setup>
const title = defineModel('title')
</script>

<template>
  <input type="text" v-model="title" />
</template>



//之前的用法
```

```
<!-- MyComponent.vue -->
<script setup>
defineProps({
  title: {
    required: true
  }
})
defineEmits(['update:title'])
</script>

<template>
  <input
    type="text"
    :value="title"
    @input="$emit('update:title', $event.target.value)"
  />
</template>
```

3. 多个v-modal绑定

```
<UserName
  v-model:first-name="first"
  v-model:last-name="last"
/>


<script setup>
const firstName = defineModel('firstName')
const lastName = defineModel('lastName')
</script>

<template>
  <input type="text" v-model="firstName" />
  <input type="text" v-model="lastName" />
</template>



====>
  <script setup>
defineProps({
  firstName: String,
  lastName: String
})

defineEmits(['update:firstName', 'update:lastName'])
</script>

<template>
  <input
    type="text"
    :value="firstName"
    @input="$emit('update:firstName', $event.target.value)"
  />
  <input
```

```
    type="text"
    :value="lastName"
    @input="$emit('update:lastName', $event.target.value)"
  />
</template>
```

4. 自定义修饰符

```
<MyComponent v-model.capitalize="myText" />

<script setup>
const [model, modifiers] = defineModel({
  set(value) {
    if (modifiers.capitalize) {
      return value.charAt(0).toUpperCase() + value.slice(1)
    }
    return value
  }
})
</script>

<template>
  <input type="text" v-model="model" />
</template>
```

5. 带参数的修饰符

```
<UserName
  v-model:first-name.capitalize="first"
  v-model:last-name.uppercase="last"
/>


<script setup>
const [firstName, firstNameModifiers] = defineModel('firstName')
const [lastName, lastNameModifiers] = defineModel('lastName')

console.log(firstNameModifiers) // { capitalize: true }
console.log(lastNameModifiers) // { uppercase: true }
</script>


=======>3.4之前
<script setup>
const props = defineProps({
firstName: String,
lastName: String,
firstNameModifiers: { default: () => ({}) },
lastNameModifiers: { default: () => ({}) }
})
defineEmits(['update:firstName', 'update:lastName'])

console.log(props.firstNameModifiers) // { capitalize: true }
console.log(props.lastNameModifiers) // { uppercase: true }
```

```
</script>
```

透传 Attributes

“透传 attribute”指的是传递给一个组件，却没有被该组件声明为 props 或 emits 的 attribute 或者 `v-on` 事件监听器。最常见的例子就是 `class`、`style` 和 `id`。

1. 如果子组件里面的根元素有了class或者style，有传入了class或者style会进行合并
2. v-on也适用于这个规则：同样的，如果原生 `button` 元素自身也通过 `v-on` 绑定了一个事件监听器，则这个监听器和从父组件继承的监听器都会被触发
3. 深层组件继承
4. 禁用继承

```
<script setup>
defineOptions({
  inheritAttrs: false
})
// ...setup 逻辑
</script>
```

这些透传进来的 attribute 可以在模板的表达式中直接用 `$attrs` 访问到。

这个 `$attrs` 对象包含了除组件所声明的 `props` 和 `emits` 之外的所有其他 attribute，例如 `class`，`style`，`v-on` 监听器等等。

- 和 props 有所不同，透传 attributes 在 JavaScript 中保留了它们原始的大小写，所以像 `foo-bar` 这样的一个 attribute 需要通过 `$attrs['foo-bar']` 来访问。
- 像 `@click` 这样的一个 `v-on` 事件监听器将在此对象下被暴露为一个函数 `$attrs.onClick`。

我们想要所有像 `class` 和 `v-on` 监听器这样的透传 attribute 都应用在内部的 `<button>` 上而不是外层的 `<div>` 上。我们可以通过设定 `inheritAttrs: false` 和使用 `v-bind="$attrs"` 来实现：

```
<div class="btn-wrapper">
  <button class="btn" v-bind="$attrs">Click Me</button>
</div>
```

5. 多根节点继承

```
<header>...</header>
<main v-bind="$attrs">...</main>
<footer>...</footer>
```

6. js访问attribute

```
<script setup>
import { useAttrs } from 'vue'

const attrs = useAttrs()
</script>
```

需要注意的是，虽然这里的 `attrs` 对象总是反映为最新的透传 attribute，**但它并不是响应式的 (考虑到性能因素)。你不能通过侦听器去监听它的变化。**如果你需要响应性，可以使用 prop。或者你也可以使用 `onUpdated()` 使得在每次更新时结合最新的 `attrs` 执行副作用。

插槽

1. 默认插槽
2. 具名插槽
3. 条件插槽

有时你需要根据插槽是否存在来渲染某些内容。

你可以结合使用 $slots 属性与 v-if 来实现。

在下面的示例中，我们定义了一个卡片组件，它拥有三个条件插槽：header、footer 和 default。 当 header、footer 或 default 存在时，我们希望包装它们以提供额外的样式：

```
<template>
  <div class="card">
    <div v-if="$slots.header" class="card-header">
      <slot name="header" />
    </div>

    <div v-if="$slots.default" class="card-content">
      <slot />
    </div>

    <div v-if="$slots.footer" class="card-footer">
      <slot name="footer" />
    </div>
  </div>
</template>
```

4. 动态插槽名

```
<base-layout>
  <template v-slot:[dynamicSlotName]>
    ...
  </template>

  <!-- 缩写为 -->
  <template #[dynamicSlotName]>
    ...
  </template>
</base-layout>
```

5. 具名插槽

然而在某些场景下插槽的内容可能想要同时使用父组件域内和子组件域内的数据。要做到这一点，我们需要一种方法来让子组件在渲染时将一部分数据提供给插槽。

我们也确实有办法这么做！可以像对组件传递 props 那样，向一个插槽的出口上传递 attributes：

```vue
<!-- <MyComponent> 的模板 -->
<div>
  <slot :text="greetingMessage" :count="1"></slot>
</div>

<MyComponent v-slot="slotProps">
  {{ slotProps.text }} {{ slotProps.count }}
</MyComponent>
```

```vue
<MyComponent>
  <!-- 使用显式的默认插槽 -->
  <template #default="{ message }">
    <p>{{ message }}</p>
  </template>

  <template #footer>
    <p>Here's some contact info</p>
  </template>
</MyComponent>
```

6. 高级列表组件实例

```vue
<FancyList :api-url="url" :per-page="10">
  <template #item="{ body, username, likes }">
    <div class="item">
      <p>{{ body }}</p>
      <p>by {{ username }} | {{ likes }} likes</p>
    </div>
  </template>
</FancyList>

<ul>
  <li v-for="item in items">
    <slot name="item" v-bind="item"></slot>
  </li>
</ul>
```

依赖注入

1. 不会自动解包
2. 当提供 / 注入响应式的数据时，**建议尽可能将任何对响应式状态的变更都保持在供给方组件中。**这样可以确保所提供状态的声明和变更操作都内聚在同一个组件内，使其更容易维护。有的时候，我们可能需要在注入方组件中更改数据。在这种情况下，我们推荐在供给方组件内声明并提供一个更改数据的方法函数：

```vue
<!-- 在供给方组件内 -->
<script setup>
import { provide, ref } from 'vue'

const location = ref('North Pole')

function updateLocation() {
```

```
  location.value = 'South Pole'
}

provide('location', {
  location,
  updateLocation
})
</script>

<!-- 在注入方组件 -->
<script setup>
import { inject } from 'vue'

const { location, updateLocation } = inject('location')
</script>

<template>
  <button @click="updateLocation">{{ location }}</button>
</template>
```

3. 使用Symbol作为注入名

```
// keys.js
export const myInjectionKey = Symbol()
```

```
// 在供给方组件中
import { provide } from 'vue'
import { myInjectionKey } from './keys.js'

provide(myInjectionKey, { /*
  要提供的数据
*/ });
// 注入方组件
import { inject } from 'vue'
import { myInjectionKey } from './keys.js'

const injected = inject(myInjectionKey)
```

异步组件?(疑惑)

在大型项目中，我们可能需要拆分应用为更小的块，并仅在需要时再从服务器加载相关组件。Vue 提供了 `defineAsyncComponent` 方法来实现此功能。

ES 模块动态导入也会返回一个 Promise，所以多数情况下我们会将它和 `defineAsyncComponent` 搭配使用。类似 Vite 和 Webpack 这样的构建工具也支持此语法 (并且会将它们作为打包时的代码分割点)，因此我们也可以用它来导入 Vue 单文件组件：

```
import { defineAsyncComponent } from 'vue'

const AsyncComp = defineAsyncComponent(() =>
  import('./components/MyComponent.vue')
)
```

最后得到的 `AsyncComp` 是一个外层包装过的组件，**仅在页面需要它渲染时才会调用加载内部实际组件的函数**。它会将接收到的 props 和插槽传给内部组件，所以你可以使用这个异步的包装组件无缝地替换原始组件，同时实现延迟加载。

加载和错误状态

```
const AsyncComp = defineAsyncComponent({
  // 加载函数
  loader: () => import('./Foo.vue'),

  // 加载异步组件时使用的组件
  loadingComponent: LoadingComponent,
  // 展示加载组件前的延迟时间，默认为 200ms
  delay: 200,

  // 加载失败后展示的组件
  errorComponent: ErrorComponent,
  // 如果提供了一个 timeout 时间限制，并超时了
  // 也会显示这里配置的报错组件，默认值是：Infinity
  timeout: 3000
})
```

suspense

Suspense 是一种用于处理异步组件和延迟加载的机制，它使得在等待异步操作完成时，可以显示一些占位内容，以提供更好的用户体验。

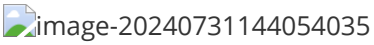
异步依赖

1. setup的await
2. 异步组件

使用

```
<RouterView v-slot="{ Component }">
  <template v-if="Component">
    <Transition mode="out-in">
      <KeepAlive>
        <Suspense>
          <!-- 主要内容 -->
          <component :is="Component"></component>

          <!-- 加载中状态 -->
          <template #fallback>
            正在加载...
          </template>
        </Suspense>
      </KeepAlive>
    </Transition>
  </template>
</RouterView>
```



逻辑复用(组件和下面的)

组合式函数

例如

```
<script setup>
import { ref, onMounted, onUnmounted } from 'vue'

const x = ref(0)
const y = ref(0)

function update(event) {
  x.value = event.pageX
  y.value = event.pageY
}

onMounted(() => window.addEventListener('mousemove', update))
onUnmounted(() => window.removeEventListener('mousemove', update))
</script>

<template>Mouse position is at: {{ x }}, {{ y }}</template>
```

抽离出来

```
// mouse.js
import { ref, onMounted, onUnmounted } from 'vue'

// 按照惯例，组合式函数名以“use”开头
export function useMouse() {
  // 被组合式函数封装和管理的状态
  const x = ref(0)
  const y = ref(0)

  // 组合式函数可以随时更改其状态。
  function update(event) {
    x.value = event.pageX
    y.value = event.pageY
  }

  // 一个组合式函数也可以挂靠在所属组件的生命周期上
  // 来启动和卸载副作用
  onMounted(() => window.addEventListener('mousemove', update))
  onUnmounted(() => window.removeEventListener('mousemove', update))

  // 通过返回值暴露所管理的状态
  return { x, y }
}
```

使用

```vue
<script setup>
import { useMouse } from './mouse.js'

const { x, y } = useMouse()
</script>

<template>Mouse position is at: {{ x }}, {{ y }}</template>
```

封装DOM事件监听器

```vue
// event.js
import { onMounted, onUnmounted } from 'vue'

export function useEventListener(target, event, callback) {
  // 如果你想的话，
  // 也可以用字符串形式的 CSS 选择器来寻找目标 DOM 元素
  onMounted(() => target.addEventListener(event, callback))
  onUnmounted(() => target.removeEventListener(event, callback))
}
```

接收响应式变化

我们可以用 `watchEffect()` 和 `toValue()` API 来重构我们现有的实现：

```vue
// fetch.js
import { ref, watchEffect, toValue } from 'vue'

export function useFetch(url) {
  const data = ref(null)
  const error = ref(null)

  const fetchData = () => {
    // reset state before fetching..
    data.value = null
    error.value = null

    fetch(toValue(url))
      .then((res) => res.json())
      .then((json) => (data.value = json))
      .catch((err) => (error.value = err))
  }

  watchEffect(() => {
    fetchData()
  })

  return { data, error }
}
```

`toValue()` 是一个在 3.3 版本中新增的 API。它的设计目的是将 ref 或 getter 规范化为值。**如果参数是 ref，它会返回 ref 的值；如果参数是函数，它会调用函数并返回其返回值。**否则，它会原样返回参数。它的工作方式类似于 `unref()`，但对函数有特殊处理。

如果你的组合式函数在输入参数是 ref 或 getter 的情况下创建了响应式 effect，为了让它能够被正确追踪，请确保要么使用 `watch()` 显式地监视 ref 或 getter，要么在 `watchEffect()` 中调用 `toValue()`

与其他模式的区别

Mixin

1. **不清晰的数据来源**：当使用了多个 mixin 时，实例上的数据属性来自哪个 mixin 变得不清晰，这使追溯实现和理解组件行为变得困难。这也是我们推荐在组合式函数中使用 ref + 解构模式的理由：让属性的来源在消费组件时一目了然。
2. **命名空间冲突**：多个来自不同作者的 mixin 可能会注册相同的属性名，造成命名冲突。若使用组合式函数，你可以通过在解构变量时对变量进行重命名来避免相同的键名。
3. **隐式的跨 mixin 交流**：多个 mixin 需要依赖共享的属性名来进行相互作用，这使得它们隐性地耦合在一起。**而一个组合式函数的返回值可以作为另一个组合式函数的参数被传入，像普通函数那样。**

无渲染组件

无渲染组件是一个不需要渲染任何自己的HTML的组件。相反，它只管理状态和行为。它会暴露一个单独的作用域，让父组件或消费者完全控制应该渲染的内容。

组合式函数相对于无渲染组件的主要优势是：**组合式函数不会产生额外的组件实例开销。当在整个应用中使用时，由无渲染组件产生的额外组件实例会带来无法忽视的性能开销。**

我们推荐在纯逻辑复用时使用组合式函数，在需要同时复用逻辑和视图布局时使用无渲染组件

React Hooks

Vue 的组合式函数是基于 Vue 细粒度的响应性系统，这和 React hooks 的执行模型有本质上的不同。

自定义指令

组件内定义

```
<script setup>
// 在模板中启用 v-focus
const vFocus = {
  mounted: (el) => el.focus()
}
</script>

<template>
  <input v-focus />
</template>
```

全局定义

```
const app = createApp({})

// 使 v-focus 在所有组件中都可用
app.directive('focus', {
  /* ... */
})
```

钩子

```
const myDirective = {
  // 在绑定元素的 attribute 前
  // 或事件监听器应用前调用
  created(el, binding, vnode) {
    // 下面会介绍各个参数的细节
  },
  // 在元素被插入到 DOM 前调用
  beforeMount(el, binding, vnode) {},
  // 在绑定元素的父组件
  // 及他自己的所有子节点都挂载完成后调用
  mounted(el, binding, vnode) {},
  // 绑定元素的父组件更新前调用
  beforeUpdate(el, binding, vnode, prevVnode) {},
  // 在绑定元素的父组件
  // 及他自己的所有子节点都更新后调用
  updated(el, binding, vnode, prevVnode) {},
  // 绑定元素的父组件卸载前调用
  beforeUnmount(el, binding, vnode) {},
  // 绑定元素的父组件卸载后调用
  unmounted(el, binding, vnode) {}
}
```

简化使用

```
app.directive('color', (el, binding) => {
  // 这会在 `mounted` 和 `updated` 时都调用
  el.style.color = binding.value
})
```

字面量传入

```
<div v-demo="{ color: 'white', text: 'hello!' }"></div>

app.directive('demo', (el, binding) => {
  console.log(binding.value.color) // => "white"
  console.log(binding.value.text) // => "hello!"
})
```

插件

插件 (Plugins) 是一种能为 Vue 添加全局功能的工具代码，例如vue-router就是vue里面的插件，vuex

编写插件

```
// plugins/i18n.js
export default {
  install: (app, options) => {
    // 注入一个全局可用的 $translate() 方法
    app.config.globalProperties.$translate = (key) => {
      // 获取 `options` 对象的深层属性
      // 使用 `key` 作为索引
      return key.split('.').reduce((o, i) => {
        if (o) return o[i]
      }, options)
    }
  }
}
```

```
import i18nPlugin from './plugins/i18n'

app.use(i18nPlugin, {
  greetings: {
    hello: 'Bonjour!'
  }
})
```

使用provide和inject

```
// plugins/i18n.js
export default {
  install: (app, options) => {
    app.provide('i18n', options)
  }
}
```

```
<script setup>
import { inject } from 'vue'

const i18n = inject('i18n')

console.log(i18n.greetings.hello)
</script>
```

源码

```
// 应用api use，使用各种plugin
  use(plugin: Plugin, ...options: any[]) {
    if (installedPlugins.has(plugin)) {
      __DEV__ && warn(`Plugin has already been applied to target app.`)
    } else if (plugin && isFunction(plugin.install)) {
        //如果插件存在并且是一个函数（isFunction(plugin.install)），则将插件添加到
installedPlugins 集合中，然后调用插件的 install 方法，并将应用程序实例（app）和其他选项传递
给它。
      installedPlugins.add(plugin)
      plugin.install(app, ...options)
    } else if (isFunction(plugin)) {
```

```
      installedPlugins.add(plugin)
      plugin(app, ...options)
    } else if (__DEV__) {
      warn(
        `A plugin must either be a function or an object with an "install" ` +
          `function.`
      )
    }
      //返回应用程序实例来方便链式调用
    return app
  },
```

内置API

面试

为什么组件里的data是个函数而不是对象呢

- 根实例对象`data`可以是对象也可以是函数（根实例是单例），不会产生数据污染情况
- 组件实例对象`data`必须为函数，目的是为了防止多个组件实例对象之间共用一个`data`，产生数据污染。采用函数的形式，`initData`时会将其作为工厂函数都会返回全新`data`对象