南京理工大学泰州科技学院

毕业设计(论文)外文资料翻译

学院 (系)：计算机科学与工程学院

专　　业：信息管理与信息系统

姓　　名：高杰

学　　号：1909120114

外文出处：Vue3.Koa2

附　　件：1. 外文资料翻译译文；2. 外文原文。

指导教师评语： 翻译内容与所做毕业设计选题贴近，译文语句通顺，语义正确，达到毕业设计翻译要求。 签名： 年　　月　　日

注：请将该封面与附件装订成册。

附件 1：外文资料翻译译文

什么是 Vue？

Vue（发音为 /vju:/，类似 view）是一款用于构建用户界面的 JavaScript 框架。它基于标准 HTML、CSS 和 JavaScript 构建，并提供了一套声明式的、组件化的编程模型，帮助你高效地开发用户界面。无论是简单还是复杂的界面，Vue 都可以胜任。

声明式渲染：Vue 基于标准 HTML 拓展了一套模板语法，使得我们可以声明式地描述最终输出的 HTML 和 JavaScript 状态之间的关系。响应性：Vue 会自动跟踪 JavaScript 状态并在其发生变化时响应式地更新 DOM。你可能已经有了些疑问——先别急，在后续的文档中我们会详细介绍每一个细节。现在，请继续看下去，以确保你对 Vue 作为一个框架到底提供了什么有一个宏观的了解。

Vue 是一个框架，也是一个生态。其功能覆盖了大部分前端开发常见的需求。但 Web 世界是十分多样化的，不同的开发者在 Web 上构建的东西可能在形式和规模上会有很大的不同。考虑到这一点，Vue 的设计非常注重灵活性和“可以被逐步集成”这个特点。根据你的需求场景，你可以用不同的方式使用 Vue：

（1）无需构建步骤，渐进式增强静态的 HTML

（2）在任何页面中作为 Web Components 嵌入

（3）单页应用 (SPA)

（4）全栈 / 服务端渲染 (SSR)

（5）Jamstack / 静态站点生成 (SSG)

（6）开发桌面端、移动端、WebGL，甚至是命令行终端中的界面

如果你是初学者，可能会觉得这些概念有些复杂。别担心！理解教程和指南的内容只需要具备基础的 HTML 和 JavaScript 知识。即使你不是这些方面的专家，也能够跟得上。

如果你是有经验的开发者，希望了解如何以最合适的方式在项目中引入 Vue，或者是对上述的这些概念感到好奇，我们在使用 Vue 的多种方式中讨论了有关它们的更多细节。

无论再怎么灵活，Vue 的核心知识在所有这些用例中都是通用的。即使你现在只是一个初学者，随着你的不断成长，到未来有能力实现更复杂的项目时，这一路上获得的知识依然会适用。如果你已经是一个老手，你可以根据实际场景来选择使用 Vue 的最佳方式，在各种场景下都可以保持同样的开发效率。这就是为什么我们将 Vue 称为“渐进式框架”：它是一个可以与你共同成长、适应你不同需求的框架。

在大多数启用了构建工具的 Vue 项目中，我们可以使用一种类似 HTML 格式的文件来书写 Vue 组件，它被称为单文件组件 （也被称为 *.vue 文件，英文 Single-File Components，缩写为 SFC)。顾名思义，Vue 的单文件组件会将一个组件的逻辑 (JavaScript)，模板 (HTML) 和样式 (CSS) 封装在同一个文件里。

Vue 的组件可以按两种不同的风格书写：选项式 API 和组合式 API。选项式 API (Options API) 使用选项式 API，我们可以用包含多

个选项的对象来描述组件的逻辑，例如 data、methods 和 mounted。选项所定义的属性都会暴露在函数内部的 this 上，它会指向当前的组件实例。

组合式 API (Composition API) 通过组合式 API，我们可以使用导入的 API 函数来描述组件逻辑。在单文件组件中，组合式 API 通常会与 <script setup> 搭配使用。这个 setup attribute 是一个标识，告诉 Vue 需要在编译时进行一些处理，让我们可以更简洁地使用组合式 API。比如，<script setup> 中的导入和顶层变量/函数都能够在模板中直接使用。

两种 API 风格都能够覆盖大部分的应用场景。它们只是同一个底层系统所提供的两套不同的接口。实际上，选项式 API 是在组合式 API 的基础上实现的！关于 Vue 的基础概念和知识在它们之间都是通用的。

选项式 API 以“组件实例”的概念为中心 (即上述例子中的 this)，对于有面向对象语言背景的用户来说，这通常与基于类的心智模型更为一致。同时，它将响应性相关的细节抽象出来，并强制按照选项来组织代码，从而对初学者而言更为友好。组合式 API 的核心思想是直接在函数作用域内定义响应式状态变量，并将从多个函数中得到的状态组合起来处理复杂问题。这种形式更加自由，也需要你对 Vue 的响应式系统有更深的理解才能高效使用。相应的，它的灵活性也使得组织和重用逻辑的模式变得更加强大。

我们相信在 Web 的世界里没有一种方案可以解决所有问题。正因如此，Vue 被设计成一个灵活的、可以渐进式集成的框架。根据使用场景的

不同需要，相应地有多种不同的方式来使用 Vue，以此在技术栈复杂度、开发体验和性能表现间取得最佳平衡。

Vue 可以以一个单独 JS 文件的形式使用，无需构建步骤！如果你的后端框架已经渲染了大部分的 HTML，或者你的前端逻辑并不复杂，不需要构建步骤，这是最简单的使用 Vue 的方式。在这些场景中你可以将 Vue 看作一个更加声明式的 jQuery 替代品。Vue 也提供了另一个更适用于此类无构建步骤场景的版本 petite-vue。它为渐进式增强已有的 HTML 作了特别的优化，功能更加精简，十分轻量。

你可以用 Vue 来构建标准的 Web Component，这些 Web Component 可以嵌入到任何 HTML 页面中，无论它们是如何被渲染的。这个方式让你能够在不需要顾虑最终使用场景的情况下使用 Vue：因为生成的 Web Component 可以嵌入到旧应用、静态 HTML，甚至用其他框架构建的应用中。一些应用在前端需要具有丰富的交互性、较深的会话和复杂的状态逻辑。构建这类应用的最佳方法是使用这样一种架构：Vue 不仅控制整个页面，还负责处理抓取新数据，并在无需重新加载的前提下处理页面切换。这种类型的应用通常称为单页应用 (Single-Page application，缩写为 SPA)。

Vue 提供了核心功能库和全面的工具链支持，为现代 SPA 提供了极佳的开发体验，覆盖以下方面：

（1）客户端路由

（2）极其快速的构建工具

（3）IDE 支持

（4）浏览器开发工具

（5）TypeScript 支持

（6）测试工具

SPA 一般要求后端提供 API 数据接口，但你也可以将 Vue 和如 Inertia.js 之类的解决方案搭配使用，在保留侧重服务端的开发模型的同时获得 SPA 的益处。纯客户端的 SPA 在首屏加载和 SEO 方面有显著的问题，因为浏览器会收到一个巨大的 HTML 空页面，只有等到 JavaScript 加载完毕才会渲染出内容。Vue 提供了一系列 API，支持将一个 Vue 应用在服务端渲染成 HTML 字符串。这能让服务器直接返回渲染好的 HTML，让用户在 JavaScript 下载完毕前就看到页面内容。Vue 之后会在客户端对应用进行“激活 (hydrate)”使其重获可交互性。这被称为服务端渲染 (SSR)，它能够极大地改善应用在 Web 核心指标上的性能表现，如最大内容绘制 (LCP)。

Vue 生态中有一些针对此类场景的、基于 Vue 的上层框架，比如 NuxtJS，能让你用 Vue 和 JavaScript 开发一个全栈应用。如果所需的数据是静态的，那么服务端渲染可以提前完成。这意味着我们可以将整个应用预渲染为 HTML，并将其作为静态文件部署。这增强了站点的性能表现，也使部署变得更容易，因为我们无需根据请求动态地渲染页面。Vue 仍可通过激活在客户端提供交互。这一技术通常被称为静态站点生成 (SSG)，也被称为 JAMStack。

SSG 有两种风格：单页和多页。这两种风格都能将站点预渲染为静态 HTML，区别在于：单页 SSG 在初始页面加载后将其“激活”为 SPA。这

需要更多的前期 JS 加载和激活成本，但后续的导航将更快，因为它只需要部分地更新页面内容，而无需重新加载整个页面。

多页 SSG 每次导航都会加载一个新页面。好处是它可以仅需最少的 JS——或者如果页面无需交互则根本不需要 JS！一些多页面 SSG 框架，如 Astro 也支持“部分激活”——它允许你通过 Vue 组件在静态 HTML 中创建交互式的“孤岛”。单页 SSG 更适合于重交互、深会话的场景，或需要在导航之间持久化元素或状态。否则，多页 SSG 将是更好的选择。

Vue 团队也维护了一个名为 VitePress 的静态站点生成器，你正在阅读的文档就是基于它构建的！VitePress 支持两种形式的 SSG。另外，NuxtJS 也支持 SSG。你甚至可以在同一个 Nuxt 应用中通过不同的路由提供 SSR 和 SSG。

尽管 Vue 主要是为构建 Web 应用而设计的，但它绝不仅仅局限于浏览器。你还可以：配合 Electron 或 Tauri 构建桌面应用配合 Ionic Vue 构建移动端应用使用 Quasar 用同一套代码同时开发桌面端和移动端应用使用 Vue 的自定义渲染 API 来构建不同目标的渲染器，比如 WebGL 甚至是终端命令行！

koa 是由 Express 原班人马打造的，致力于成为一个更小、更富有表现力、更健壮的 Web 框架。使用 koa 编写 web 应用，通过组合不同的 generator，可以免除重复繁琐的回调函数嵌套，并极大地提升错误处理的效率。koa 不在内核方法中绑定任何中间件，它仅仅提供了一个轻量优雅的函数库，使得编写 Web 应用变得得心应手。

Koa 应用是一个包含一系列中间件 generator 函数的对象。 这些中间件函数基于 request 请求以一个类似于栈的结构组成并依次执行。Koa 类似于其他中间件系统（比如 Ruby's Rack 、Connect 等）， 然而 Koa 的核心设计思路是为中间件层提供高级语法糖封装，以增强其互用性和健壮性，并使得编写中间件变得相当有趣。Koa 包含了像 content-negotiation（内容协商）、cache freshness（缓存刷新）、proxy support（代理支持）和 redirection（重定向）等常用任务方法。 与提供庞大的函数支持不同，Koa 只包含很小的一部分，因为 Koa 并不绑定任何中间件。Koa 的中间件通过一种更加传统（您也许会很熟悉）的方式进行级联，摒弃了以往 node 频繁的回调函数造成的复杂代码逻辑。 然而，使用异步函数，我们可以实现"真正" 的中间件。与之不同，当执行到 yield next 语句时，Koa 暂停了该中间件，继续执行下一个符合请求的中间件('downstrem')，然后控制权再逐级返回给上层中间件('upstream')。

下面的例子在页面中返回 "Hello World"，然而当请求开始时，请求先经过 x-response-time 和 logging 中间件，并记录中间件执行起始时间。 然后将控制权交给 reponse 中间件。当一个中间件调用 next() 函数时，函数挂起并控件传递给定义的下一个中间件。在没有更多的中间件执行下游之后，堆栈将退出，并且每个中间件被恢复以执行其上游行为。应用配置是 app 实例属性，目前支持的配置项如下：app.env 默认

为 NODE_ENV or "development"，app.proxy 如果为 true，则解析 "Host" 的 header 域。

app.subdomainOffset 默认为 2，表示 .subdomains 所忽略的字符偏移量。app.listen(...)，Koa 应用并非是一个 1-to-1 表征关系的 HTTP 服务器。 一个或多个 Koa 应用可以被挂载到一起组成一个包含单一 HTTP 服务器的大型应用群。如下为一个绑定 3000 端口的简单 Koa 应用，其创建并返回了一个 HTTP 服务器，为 Server#listen() 传递指定参数 app.callback()返回一个适合 http.createServer() 方法的回调函数用来处理请求。 您也可以使用这个回调函数将您的 app 挂载在 Connect/Express 应用上。app.context 是从中创建 ctx 的原型。 可以通过编辑 app.context 向 ctx 添加其他属性。当需要将 ctx 添加到整个应用程序中使用的属性或方法时，这将会非常有用。这可能会更加有效（不需要中间件）和/或更简单（更少的 require()），而不必担心更多的依赖于 ctx，这可以被看作是一种反向模式。默认情况下 Koa 会将所有错误信息输出到 stderr， 除非 app.silent 是 true.当 err.status 是 404 或者 err.expose 时，默认错误处理程序也不会输出错误。Context(上下文)Koa Context 将 node 的 request 和 response 对象封装在一个单独的对象里面，其为编写 web 应用和 API 提供了很多有用的方法。 这些操作在 HTTP 服务器开发中经常使用，因此其被添加在上下文这一层，而不是更高层框架中，因此将迫使中间件需要重新实现这些常用方法。 Koa Request 对象是对 node 的 request 进一步抽象和封装，提供了日常 HTTP 服务器开发中一些有用的功能。Koa Response 对象是对

node 的 response 进一步抽象和封装，提供了日常 HTTP 服务器开发中一些有用的功能。

附件 2：外文原文

ⓘ You are reading the documentation for Vue 3!

- Vue 2 support will end on Dec 31, 2023. Learn more about Vue 2 Extended LTS.
- Vue 2 documentation has been moved to v2.vuejs.org.
- Upgrading from Vue 2? Check out the Migration Guide.

Learn Vue with video tutorials on VueMastery.com

What is Vue?

Vue (pronounced /vju:/, like **view**) is a JavaScript framework for building user interfaces. It builds on top of standard HTML, CSS, and JavaScript and provides a declarative and component-based programming model that helps you efficiently develop user interfaces, be they simple or complex.

Here is a minimal example:

interfaces, be they simple or complex.

Here is a minimal example:

```js
import { createApp } from 'vue'

createApp({
  data() {
    return {
      count: 0
    }
  }
}).mount('#app')
```

```
<div id="app">
  <button @click="count++">
    Count is: {{ count }}
  </button>
</div>
```

Result

Count is: 0

The above example demonstrates the two core features of Vue:

Count is: 0

The above example demonstrates the two core features of Vue:

- **Declarative Rendering**: Vue extends standard HTML with a template syntax that allows us to declaratively describe HTML output based on JavaScript state.
- **Reactivity**: Vue automatically tracks JavaScript state changes and efficiently updates the DOM when changes happen.

You may already have questions - don't worry. We will cover every little detail in the rest of the documentation. For now, please read along so you can have a high-level understanding of what Vue offers.

Prerequisites

The rest of the documentation assumes basic familiarity with HTML, CSS, and JavaScript. If you are totally new to frontend development, it might not be the best idea to jump right into a framework as your first step - grasp the basics and then come back! You can check your knowledge level with this JavaScript overview. Prior experience with other frameworks helps, but is not required.

The Progressive Framework

Vue is a framework and ecosystem that covers most of the common features needed in frontend development. But the web is extremely diverse - the things we build on the web may vary drastically in form and scale. With that in mind, Vue is designed to be flexible and incrementally adoptable. Depending on your use case, Vue can be used in different ways:

- Enhancing static HTML without a build step
- Embedding as Web Components on any page
- Single-Page Application (SPA)
- Fullstack / Server-Side Rendering (SSR)
- Jamstack / Static Site Generation (SSG)
- Targeting desktop, mobile, WebGL, and even the terminal

If you find these concepts intimidating, don't worry! The tutorial and guide only require basic HTML and JavaScript knowledge, and you should be able to follow along without being an expert in any of these.

If you are an experienced developer interested in how to best integrate Vue into your stack, or you are curious about what these terms mean, we discuss them in more detail in Ways of Using Vue.

If you are an experienced developer interested in how to best integrate Vue into your stack, or you are curious about what these terms mean, we discuss them in more detail in Ways of Using Vue.

Despite the flexibility, the core knowledge about how Vue works is shared across all these use cases. Even if you are just a beginner now, the knowledge gained along the way will stay useful as you grow to tackle more ambitious goals in the future. If you are a veteran, you can pick the optimal way to leverage Vue based on the problems you are trying to solve, while retaining the same productivity. This is why we call Vue "The Progressive Framework": it's a framework that can grow with you and adapt to your needs.

Single-File Components

In most build-tool-enabled Vue projects, we author Vue components using an HTML-like file format called **Single-File Component** (also known as `*.vue` files, abbreviated as **SFC**). A Vue SFC, as the name suggests, encapsulates the component's logic (JavaScript), template (HTML), and styles (CSS) in a single file. Here's the previous example, written in SFC format:

example, written in SFC format:

```
<script>
export default {
  data() {
    return {
      count: 0
    }
  }
}
</script>

<template>
  <button @click="count++">Count is: {{ count }}</button>
</template>

<style scoped>
button {
  font-weight: bold;
}
</style>
```

SFC is a defining feature of Vue and is the recommended way to author Vue components **if** your use case warrants a build setup. You can learn more about the how and why of SFC in its dedicated section - but for now, just know that Vue will handle all the build tools setup for you.

API Styles

Vue components can be authored in two different API styles: **Options API** and **Composition API**.

Options API

With Options API, we define a component's logic using an object of options such as `data`, `methods`, and `mounted`. Properties defined by options are exposed on `this` inside functions, which points to the component instance:

```
<script>
export default {
  // Properties returned from data() become reactive state
  // and will be exposed on `this`.
  data() {
    return {
      count: 0
    }
  },

  // Methods are functions that mutate state and trigger updates.
  // They can be bound as event listeners in templates.
  methods: {
```

```
    // This function will be called when the component is mounted.
    mounted() {
      console.log(`The initial count is ${this.count}.`)
    }
  }
</script>

<template>
  <button @click="increment">Count is: {{ count }}</button>
</template>
```

Try it in the Playground

Composition API

With Composition API, we define a component's logic using imported API functions. In SFCs, Composition API is typically used with `<script setup>`. The `setup` attribute is a hint that makes Vue perform compile-time transforms that allow us to use Composition API with less boilerplate. For example, imports and top-level variables / functions declared in `<script setup>` are directly usable in the template.

Here is the same component, with the exact same template, but using Composition API and `<script setup>` instead:

Here is the same component, with the exact same template, but using Composition API and `<script setup>` instead:

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

// reactive state
const count = ref(0)

// functions that mutate state and trigger updates
function increment() {
  count.value++
}

// lifecycle hooks
onMounted(() => {
  console.log(`The initial count is ${count.value}.`)
})
</script>

<template>
  <button @click="increment">Count is: {{ count }}</button>
</template>
```

Try it in the Playground

Which to Choose?

Both API styles are fully capable of covering common use cases. They are different interfaces powered by the exact same underlying system. In fact, the Options API is implemented on top of the Composition API! The fundamental concepts and knowledge about Vue are shared across the two styles.

The Options API is centered around the concept of a "component instance" (`this` as seen in the example), which typically aligns better with a class-based mental model for users coming from OOP language backgrounds. It is also more beginner-friendly by abstracting away the reactivity details and enforcing code organization via option groups.

The Composition API is centered around declaring reactive state variables directly in a function scope and composing state from multiple functions together to handle complexity. It is more free-form and requires an understanding of how reactivity works in Vue to be used effectively. In return, its flexibility enables more powerful patterns for organizing and reusing logic.

You can learn more about the comparison between the two styles and the potential benefits of Composition API in the Composition API FAQ.

If you are new to Vue, here's our general recommendation:

- For learning purposes, go with the style that looks easier to understand to you. Again, most of the core concepts are shared between the two styles. You can always

If you are new to Vue, here's our general recommendation:

- For learning purposes, go with the style that looks easier to understand to you. Again, most of the core concepts are shared between the two styles. You can always pick up the other style later.
- For production use:
  - Go with Options API if you are not using build tools, or plan to use Vue primarily in low-complexity scenarios, e.g. progressive enhancement.
  - Go with Composition API + Single-File Components if you plan to build full applications with Vue.

You don't have to commit to only one style during the learning phase. The rest of the documentation will provide code samples in both styles where applicable, and you can toggle between them at any time using the **API Preference switches** at the top of the left sidebar.