4. Vue Router & SSR

1. 课程资料

Vue Router & SSR.zip
10.86MB

2. 课程目标

- 初级：
  - 掌握 Vue Router 的基础使用；
  - 掌握 CSR/SSR 的区别；
- 中级：
  - 掌握 Vue Router 的高阶用法；
  - 掌握 Vue SSR 的基础使用；
- 高级：
  - 与 React Router 进行对比，掌握前端路由生态库的发展现状；
  - 掌握前端框架 SSR 的核心原理；

3. 课程大纲

- `Vue Router` 的基础使用；
- `Vue Router` 的进阶用法；
- 什么是 `SSR`；
- `Vue SSR`；

4. Vue Router 基础使用

> github：https://router.vuejs.org/zh/

最新的 `Vue Router` 已经发展到了 V4 版本。

从发展趋势来看，后续的前端路由都会往函数式的编程方式发展（如 `useRouter`，`useRoute`）。不管是 `React Router` 还是 `Vue Router`，也都趋向于 `Hooks` 的使用。

路由的目的：将我们的组件映射到路由上，让 `Vue Router` 知道在哪里渲染它们。

```
<script src="https://unpkg.com/vue@3"></script>
<script src="https://unpkg.com/vue-router@4"></script>

<div id="app">
  <h1>Hello App!</h1>
  <p>
    <!--使用 router-link 组件进行导航 -->
    <!--通过传递 `to` 来指定链接 -->
    <!--`<router-link>` 将呈现一个带有正确 `href` 属性的 `<a>` 标签-->
    <router-link to="/">Go to Home</router-link>
    <router-link to="/about">Go to About</router-link>
  </p>
  <!-- 路由出口 -->
  <!-- 路由匹配到的组件将渲染在这里 -->
  <router-view></router-view>
</div>
```

其中：

- `router-link` 类似 `a` 标签，这使得 `Vue Router` 可以在不重新加载页面的情况下更改 URL，处理 URL 的生成以及编码；
- `router-view` 将显示与 url 对应的组件；

一个最基本的例子：

```
// 1. 定义路由组件.
// 也可以从其他文件导入
const Home = { template: '<div>Home</div>' }
const About = { template: '<div>About</div>' }

// 2. 定义一些路由
// 每个路由都需要映射到一个组件。
// 我们后面再讨论嵌套路由。
const routes = [
  { path: '/', component: Home },
  { path: '/about', component: About },
]

// 3. 创建路由实例并传递 `routes` 配置
```

```
// 你可以在这里输入更多的配置，但我们在这里
// 暂时保持简单
const router = VueRouter.createRouter({
  // 4. 内部提供了 history 模式的实现。为了简单起见，我们在这里使用 hash 模式。
  history: VueRouter.createWebHashHistory(),
  routes, // `routes: routes` 的缩写
})

// 5. 创建并挂载根实例
const app = Vue.createApp({})
//确保 _use_ 路由实例使
//整个应用支持路由。
app.use(router)

app.mount('#app')

// 现在，应用已经启动了!
```

4.1 动态参数路由

```
const User = {
  template: '<div>User</div>',
}

// 这些都会传递给 `createRouter`
const routes = [
  // 动态字段以冒号开始
  { path: '/users/:id', component: User },
]
// /users/johnny 和 /users/jolyne 这样的 URL 都会映射到同一个路由。
// 用冒号 : 表示。当一个路由被匹配时，它的 params 的值将在每个组件中以
this.$route.params 的形式暴露出来。
// 因此，我们可以通过更新 User 的模板来呈现当前的用户 ID

const User = {
  template: '<div>User {{ $route.params.id }}</div>',
}
```

使用带有参数的路由时需要注意的是，当用户从 `/users/johnny` 导航到 `/users/jolyne` 时，相同的组件实例将被重复使用。因为两个路由都渲染同个组件，比起销毁再创建，复用则显得更加高效。不过，这也意味着组件的生命周期钩子不会被调用。

要对同一个组件中参数的变化做出响应的话，你可以简单地 watch `$route` 对象上的任意属性，在这个场景中，就是 `$route.params` ：

```
const User = {
  template: '...',
  created() {
    this.$watch(
      () => this.$route.params,
      (toParams, previousParams) => {
        // 对路由变化做出响应...
      }
    )
  },
}
```

或者使用导航守卫：

```
const User = {
  template: '...',
  async beforeRouteUpdate(to, from) {
    // 对路由变化做出响应...
    this.userData = await fetchUser(to.params.id)
  },
}
```

4.2 路由的匹配语法

定义像 `:userId` 这样的参数时，我们内部使用以下的正则 `([^/]+)` (至少有一个字符不是斜杠 `/`)来从 URL 中提取参数。这很好用，除非你需要根据参数的内容来区分两个路由。想象一下，两个路由 `/:orderId` 和 `/:productName` ，两者会匹配完全相同的 URL，所以我们需要一种方法来区分它们。最简单的方法就是在路径中添加一个静态部分来区分它们：

```
const routes = [
  // 匹配 /o/3549
  { path: '/o/:orderId' },
  // 匹配 /p/books
  { path: '/p/:productName' },
]
```

但在某些情况下，我们并不想添加静态的 `/o` `/p` 部分。由于， `orderId` 总是一个数字，而 `productName` 可以是任何东西，所以我们可以在括号中为参数指定一个自定义的正则：

```
const routes = [
  // /:orderId -> 仅匹配数字
  { path: '/:orderId(\\d+)' },
  // /:productName -> 匹配其他任何内容
  { path: '/:productName' },
]
```

默认情况下，所有路由是不区分大小写的，并且能匹配带有或不带有尾部斜线的路由。例如，路由 `/users` 将匹配 `/users` 、 `/users/` 、甚至 `/Users/` 。这种行为可以通过 `strict` 和 `sensitive` 选项来修改，它们可以既可以应用在整个全局路由上，又可以应用于当前路由上：

```
const router = createRouter({
  history: createWebHistory(),
  routes: [
    // 将匹配 /users/posva 而非：
    // - /users/posva/ 当 strict: true
    // - /Users/posva 当 sensitive: true
    { path: '/users/:id', sensitive: true },
    // 将匹配 /users, /Users, 以及 /users/42 而非 /users/ 或 /users/42/
    { path: '/users/:id?' },
  ],
  strict: true, // applies to all routes
})
```

```
、、
const routes = [
  // 匹配 /users 和 /users/posva
  { path: '/users/:userId?' },
  // 匹配 /users 和 /users/42
  { path: '/users/:userId(\\d+)?' },
]
```

可以通过使用 `?` 修饰符(0 个或 1 个)将一个参数标记为可选：

```
const routes = [
  // 匹配 /users 和 /users/posva
```

```
  { path: '/users/:userId?' },
  // 匹配 /users 和 /users/42
  { path: '/users/:userId(\\d+)?' },
]
```

> 路由排序的匹配规则是基于score值来判断当前path为哪个route的，有兴趣可以先行了解：
>
> https://github.com/vuejs/router/blob/main/packages/router/src/matcher/pathParserRanker.ts#L100
>
> 或者通过以下链接测试：
>
> https://paths.esm.dev/

4.3 嵌套路由

如果我们在 `User` 组件的模板内添加一个 `<router-view>`：

```
const User = {
  template: `
    <div class="user">
      <h2>User {{ $route.params.id }}</h2>
      <router-view></router-view>
    </div>
  `,
}
```

要将组件渲染到这个嵌套的 `router-view` 中，我们需要在路由中配置 `children`：

```
const routes = [
  {
    path: '/user/:id',
    component: User,
    children: [
      {
        // 当 /user/:id/profile 匹配成功
        // UserProfile 将被渲染到 User 的 <router-view> 内部
        path: 'profile',
        component: UserProfile,
      },
      {
        // 当 /user/:id/posts 匹配成功
        // UserPosts 将被渲染到 User 的 <router-view> 内部
        path: 'posts',
```

```
        component: UserPosts,
      },
    ],
  },
]
```

注意，以 `/` 开头的嵌套路径将被视为根路径。这允许你利用组件嵌套，而不必使用嵌套的 URL。

如你所见，`children` 配置只是另一个路由数组，就像 `routes` 本身一样。因此，你可以根据自己的需要，不断地嵌套视图。

此时，按照上面的配置，当你访问 `/user/eduardo` 时，在 `User` 的 `router-view` 里面什么都不会呈现，因为没有匹配到嵌套路由。也许你确实想在那里渲染一些东西。在这种情况下，你可以提供一个空的嵌套路径：

```
const routes = [
  {
    path: '/user/:id',
    component: User,
    children: [
      // 当 /user/:id 匹配成功
      // UserHome 将被渲染到 User 的 <router-view> 内部
      { path: '', component: UserHome },

      // ...其他子路由
    ],
  },
]
```

嵌套的命名路由：

```
const routes = [
  {
    path: '/user/:id',
    component: User,
    // 请注意，只有子路由具有名称
    children: [{ path: '', name: 'user', component: UserHome }],
  },
]
```

4.4 编程式导航

在 Vue 实例中，你可以通过 `$router` 访问路由实例。因此你可以调用 `this.$router.push`。

想要导航到不同的 URL，可以使用 `router.push` 方法。这个方法会向 history 栈添加一个新的记录，所以，当用户点击浏览器后退按钮时，会回到之前的 URL。

当你点击 `<router-link>` 时，内部会调用这个方法，所以点击 `<router-link :to="...">` 相当于调用 `router.push(...)`：

```
const username = 'eduardo'
// 我们可以手动建立 url，但我们必须自己处理编码
router.push(`/user/${username}`) // -> /user/eduardo
// 同样
router.push({ path: `/user/${username}` }) // -> /user/eduardo
// 如果可能的话，使用 `name` 和 `params` 从自动 URL 编码中获益
router.push({ name: 'user', params: { username } }) // -> /user/eduardo
// `params` 不能与 `path` 一起使用
router.push({ path: '/user', params: { username } }) // -> /user
```

替换当前路由位置时：

```
router.push({ path: '/home', replace: true })
// 相当于
router.replace({ path: '/home' })
```

4.5 命名路由

除了 `path` 之外，你还可以为任何路由提供 `name`。这有以下优点：

- 没有硬编码的 URL
- `params` 的自动编码/解码。
- 防止你在 url 中出现打字错误。
- 绕过路径排序（如显示一个）

```
import Vue from 'vue'
import VueRouter from 'vue-router'

Vue.use(VueRouter)

const Home = { template: '<div>This is Home</div>' }
const Foo = { template: '<div>This is Foo</div>' }
const Bar = { template: '<div>This is Bar {{ $route.params.id }}</div>' }
```

```
const router = new VueRouter({
  mode: 'history',
  base: __dirname,
  routes: [
    { path: '/', name: 'home', component: Home },
    { path: '/foo', name: 'foo', component: Foo },
    { path: '/bar/:id', name: 'bar', component: Bar }
  ]
})

new Vue({
  router,
  template: `
    <div id="app">
      <h1>Named Routes</h1>
      <p>Current route name: {{ $route.name }}</p>
      <ul>
        <li><router-link :to="{ name: 'home' }">home</router-link></li>
        <li><router-link :to="{ name: 'foo' }">foo</router-link></li>
        <li><router-link :to="{ name: 'bar', params: { id: 123 }}">bar</router-
link></li>
      </ul>
      <router-view class="view"></router-view>
    </div>
  `
}).$mount('#app')
```

要链接到一个命名的路由，可以向 `router-link` 组件的 `to` 属性传递一个对象：

```
<router-link :to="{ name: 'user', params: { username: 'erina' }}">
  User
</router-link>
```

与代码调用 `router.push()` 是一回事：

```
router.push({ name: 'user', params: { username: 'erina' } })
```

4.6 重定向和别名

重定向也是通过 `routes` 配置来完成，下面例子是从 `/home` 重定向到 `/`：

```
const routes = [{ path: '/home', redirect: '/' }]
```

重定向的目标也可以是一个命名的路由：

```
const routes = [{ path: '/home', redirect: { name: 'homepage' } }]
```

动态返回重定向目标：

```
const routes = [
  {
    // /search/screens -> /search?q=screens
    path: '/search/:searchText',
    redirect: to => {
      // 方法接收目标路由作为参数
      // return 重定向的字符串路径/路径对象
      return { path: '/search', query: { q: to.params.searchText } }
    },
  },
  {
    path: '/search',
    // ...
  },
]
```

定位到相对重定向：

```
const routes = [
  {
    // 将总是把/users/123/posts重定向到/users/123/profile。
    path: '/users/:id/posts',
    redirect: to => {
      // 该函数接收目标路由作为参数
      // 相对位置不以`/`开头
      // 或 { path: 'profile'}
      return 'profile'
    },
  },
]
```

4.7 不同的路由模式

4.7.1 Hash模式

```
import { createRouter, createWebHashHistory } from 'vue-router'

const router = createRouter({
  history: createWebHashHistory(),
  routes: [
    //...
  ],
})
```

它在内部传递的实际 URL 之前使用了一个哈希字符（`#`）。由于这部分 URL 从未被发送到服务器，所以它不需要在服务器层面上进行任何特殊处理。不过，它在 SEO 中确实有不好的影响。如果你担心这个问题，可以使用 HTML5 模式。

4.7.2 html5模式

```
import { createRouter, createWebHistory } from 'vue-router'

const router = createRouter({
  history: createWebHistory(),
  routes: [
    //...
  ],
})
```

当使用这种历史模式时，URL 会看起来很 "正常"，例如 `https://example.com/user/id`。漂亮!

不过，问题来了。由于我们的应用是一个单页的客户端应用，如果没有适当的服务器配置，用户在浏览器中直接访问 `https://example.com/user/id`，就会得到一个 404 错误。

4.8 手写Vue Router

核心代码：

```
//myVueRouter.js
let Vue = null;
class HistoryRoute {
```

```
        constructor() {
            this.current = null;
        }
    }
    class VueRouter {
        constructor(options) {
            this.mode = options.mode || 'hash';
            this.routes = options.routes || []; //你传递的这个路由是一个数组表
            this.routesMap = this.createMap(this.routes);
            this.history = new HistoryRoute();
            this.init();
        }
        init() {
            if (this.mode === 'hash') {
                // 先判断用户打开时有没有hash值，没有的话跳转到#/
                location.hash ? '' : (location.hash = '/');
                window.addEventListener('load', () => {
                    this.history.current = location.hash.slice(1);
                });
                window.addEventListener('hashchange', () => {
                    this.history.current = location.hash.slice(1);
                });
            } else {
                location.pathname ? '' : (location.pathname = '/');
                window.addEventListener('load', () => {
                    this.history.current = location.pathname;
                });
                window.addEventListener('popstate', () => {
                    this.history.current = location.pathname;
                });
            }
        }

        createMap(routes) {
            return routes.reduce((pre, current) => {
                pre[current.path] = current.component;
                return pre;
            }, {});
        }
    }
    VueRouter.install = function (v) {
        Vue = v;
        Vue.mixin({
            beforeCreate() {
                if (this.$options && this.$options.router) {
                    // 如果是根组件
                    this._root = this; //把当前实例挂载到_root上
```

```
                this._router = this.$options.router;
                Vue.util.defineReactive(this, 'xxx', this._router.history);
            } else {
                //如果是子组件
                this._root = this.$parent && this.$parent._root;
            }
            Object.defineProperty(this, '$router', {
                get() {
                    return this._root._router;
                },
            });
            Object.defineProperty(this, '$route', {
                get() {
                    return this._root._router.history.current;
                },
            });
        },
    });
    Vue.component('router-link', {
        props: {
            to: String,
        },
        render(h) {
            let mode = this._self._root._router.mode;
            let to = mode === 'hash' ? '#' + this.to : this.to;
            return h('a', { attrs: { href: to } }, this.$slots.default);
        },
    });
    Vue.component('router-view', {
        render(h) {
            let current = this._self._root._router.history.current;
            let routeMap = this._self._root._router.routesMap;
            return h(routeMap[current]);
        },
    });
};

export default VueRouter;
```

4.9 Vue Router 进阶使用

4.9.1 导航守卫

1. 导航被触发。

2. 在失活的组件里调用 `beforeRouteLeave` 守卫。
3. 调用全局的 `beforeEach` 守卫。
4. 在重用的组件里调用 `beforeRouteUpdate` 守卫(2.2+)。
5. 在路由配置里调用 `beforeEnter`。
6. 解析异步路由组件。
7. 在被激活的组件里调用 `beforeRouteEnter`。
8. 调用全局的 `beforeResolve` 守卫(2.5+)。
9. 导航被确认。
10. 调用全局的 `afterEach` 钩子。
11. 触发 DOM 更新。
12. 调用 `beforeRouteEnter` 守卫中传给 `next` 的回调函数，创建好的组件实例会作为回调函数的参数传入。

4.9.2 全局前置守卫

你可以使用 `router.beforeEach` 注册一个全局前置守卫：

```
const router = createRouter({ ... })

router.beforeEach((to, from) => {
  // ...
  // 返回 false 以取消导航
  return false
})
```

当一个导航触发时，全局前置守卫按照创建顺序调用。守卫是异步解析执行，此时导航在所有守卫 resolve 完之前一直处于等待中。

每个守卫方法接收两个参数：

- `to`: 即将要进入的目标
- `from`: 当前导航正要离开的路由

可以返回的值如下:

- `false`: 取消当前的导航。如果浏览器的 URL 改变了(可能是用户手动或者浏览器后退按钮)，那么 URL 地址会重置到 `from` 路由对应的地址。

- 一个路由地址: 通过一个路由地址跳转到一个不同的地址，就像调用 `router.push()` 一样，你可以设置诸如 `replace: true` 或 `name: 'home'` 之类的配置。当前的导航被中断，然后进行一个新的导航，就和 `from` 一样。

```
router.beforeEach(async (to, from) => {
   if (
     // 检查用户是否已登录
     !isAuthenticated &&
     // ❗ 避免无限重定向
     to.name !== 'Login'
   ) {
     // 将用户重定向到登录页面
     return { name: 'Login' }
   }
 })
```

如果遇到了意料之外的情况，可能会抛出一个 `Error`。这会取消导航并且调用 `router.onError()` 注册过的回调。

如果什么都没有， `undefined` 或返回 `true` ，则导航是有效的，并调用下一个导航守卫

以上所有都同 `async` 函数 和 Promise 工作方式一样：

```
router.beforeEach(async (to, from) => {
  // canUserAccess() 返回 `true` 或 `false`
  const canAccess = await canUserAccess(to)
  if (!canAccess) return '/login'
})
```

4.9.3 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，不同的是，解析守卫刚好会在导航被确认之前、所有组件内守卫和异步路由组件被解析之后调用。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
      }
    }
  }
})
```

4.9.4 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)
})
```

4.9.5 路由独享守卫

你可以直接在路由配置上定义 `beforeEnter` 守卫：

```
const routes = [
  {
    path: '/users/:id',
    component: UserDetails,
    beforeEnter: (to, from) => {
      // reject the navigation
      return false
    },
  },
]
```

`beforeEnter` 守卫 只在进入路由时触发，不会在 `params`、`query` 或 `hash` 改变时触发。例如，从 `/users/2` 进入到 `/users/3` 或者从 `/users/2#info` 进入到 `/users/2#projects`。它们只有在 从一个不同的 路由导航时，才会被触发。

4.9.6 组件内的守卫

你可以为路由组件添加以下配置：

- `beforeRouteEnter`

- `beforeRouteUpdate`
- `beforeRouteLeave`

```
const UserDetails = {
  template: `...`,
  beforeRouteEnter(to, from) {
    // 在渲染该组件的对应路由被验证前调用
    // 不能获取组件实例 `this` !
    // 因为当守卫执行时，组件实例还没被创建!
  },
  beforeRouteUpdate(to, from) {
    // 在当前路由改变，但是该组件被复用时调用
    // 举例来说，对于一个带有动态参数的路径 `/users/:id`，在 `/users/1` 和
`/users/2` 之间跳转的时候，
    // 由于会渲染同样的 `UserDetails` 组件，因此组件实例会被复用。而这个钩子就会在这个情
况下被调用。
    // 因为在这种情况发生的时候，组件已经挂载好了，导航守卫可以访问组件实例 `this`
  },
  beforeRouteLeave(to, from) {
    // 在导航离开渲染该组件的对应路由时调用
    // 与 `beforeRouteUpdate` 一样，它可以访问组件实例 `this`
  },
}
```

4.9.7 数据获取

有时候，进入某个路由后，需要从服务器获取数据。例如，在渲染用户信息时，你需要从服务器获取用户的数据。我们可以通过两种方式来实现：

- 导航完成之后获取：先完成导航，然后在接下来的组件生命周期钩子中获取数据。在数据获取期间显示“加载中”之类的指示。
- 导航完成之前获取：导航完成前，在路由进入的守卫中获取数据，在数据获取成功后执行导航。

4.9.7.1 导航完成后获取数据

```
<template>
  <div class="post">
    <div v-if="loading" class="loading">Loading...</div>

    <div v-if="error" class="error">{{ error }}</div>

    <div v-if="post" class="content">
      <h2>{{ post.title }}</h2>
```

```
      <p>{{ post.body }}</p>
    </div>
  </div>
</template>

export default {
  data() {
    return {
      loading: false,
      post: null,
      error: null,
    }
  },
  created() {
    // watch 路由的参数，以便再次获取数据
    this.$watch(
      () => this.$route.params,
      () => {
        this.fetchData()
      },
      // 组件创建完后获取数据，
      // 此时 data 已经被 observed 了
      { immediate: true }
    )
  },
  methods: {
    fetchData() {
      this.error = this.post = null
      this.loading = true
      // replace `getPost` with your data fetching util / API wrapper
      getPost(this.$route.params.id, (err, post) => {
        this.loading = false
        if (err) {
          this.error = err.toString()
        } else {
          this.post = post
        }
      })
    },
  },
}
```

4.9.7.2 导航完成前获取数据

```
export default {
```

```
  data() {
    return {
      post: null,
      error: null,
    }
  },
  beforeRouteEnter(to, from, next) {
    getPost(to.params.id, (err, post) => {
      next(vm => vm.setData(err, post))
    })
  },
  // 路由改变前，组件就已经渲染完了
  // 逻辑稍稍不同
  async beforeRouteUpdate(to, from) {
    this.post = null
    try {
      this.post = await getPost(to.params.id)
    } catch (error) {
      this.error = error.toString()
    }
  },
}
```

4.10 组合式API的使用

因为我们在 `setup` 里面没有访问 `this`，所以我们不能再直接访问 `this.$router` 或 `this.$route`。作为替代，我们使用 `useRouter` 和 `useRoute` 函数：

> github地址：https://github.com/vuejs/router/blob/main/packages/router/src/useApi.ts

```
import { useRouter, useRoute } from 'vue-router'

export default {
  setup() {
    const router = useRouter()
    const route = useRoute()

    function pushWithQuery(query) {
      router.push({
        name: 'search',
        query: {
          ...route.query,
          ...query,
        },
      })
```

```
      }
    },
  }
```

4.11 路由懒加载

当打包构建应用时，JavaScript 包会变得非常大，影响页面加载。如果我们能把不同路由对应的组件分割成不同的代码块，然后当路由被访问的时候才加载对应组件，这样就会更加高效。

Vue Router 支持开箱即用的动态导入，这意味着你可以用动态导入代替静态导入：

```
// 将
// import UserDetails from './views/UserDetails.vue'
// 替换成
const UserDetails = () => import('./views/UserDetails.vue')

const router = createRouter({
  // ...
  routes: [{ path: '/users/:id', component: UserDetails }],
})
```

一般来说，对所有的路由都使用动态导入是个好主意。

4.12 导航故障

当使用 `router-link` 组件时，Vue Router 会自动调用 `router.push` 来触发一次导航。虽然大多数链接的预期行为是将用户导航到一个新页面，但也有少数情况下用户将留在同一页面上：

- 用户已经位于他们正在尝试导航到的页面
- 一个导航守卫通过调用 `return false` 中断了这次导航
- 当前的导航守卫还没有完成时，一个新的导航守卫会出现了
- 一个导航守卫通过返回一个新的位置，重定向到其他地方 (例如，`return '/login'`)
- 一个导航守卫抛出了一个 `Error`

4.12.1 检测导航故障

如果导航被阻止，导致用户停留在同一个页面上，由 `router.push` 返回的 `Promise` 的解析值将是 *Navigation Failure*。否则，它将是一个 *falsy* 值(通常是 `undefined`)。这样我们就可以区分我们导航是否离开了当前位置：

```
const navigationResult = await router.push('/my-profile')
```

```
if (navigationResult) {
  // 导航被阻止
} else {
  // 导航成功（包括重新导航的情况）
  this.isMenuOpen = false
}
```

Navigation Failure 是带有一些额外属性的 `Error` 实例，这些属性为我们提供了足够的信息，让我们知道哪些导航被阻止了以及为什么被阻止了。要检查导航结果的性质，请使用 `isNavigationFailure` 函数：

```
import { NavigationFailureType, isNavigationFailure } from 'vue-router'

// 试图离开未保存的编辑文本界面
const failure = await router.push('/articles/2')

if (isNavigationFailure(failure, NavigationFailureType.aborted)) {
  // 给用户显示一个小通知
  showToast('You have unsaved changes, discard and leave anyway?')
}
```

总共有三种不同的类型：

- `aborted`：在导航守卫中返回 `false` 中断了本次导航。
- `cancelled`：在当前导航还没有完成之前又有了一个新的导航。比如，在等待导航守卫的过程中又调用了 `router.push`。
- `duplicated`：导航被阻止，因为我们已经在目标位置了。

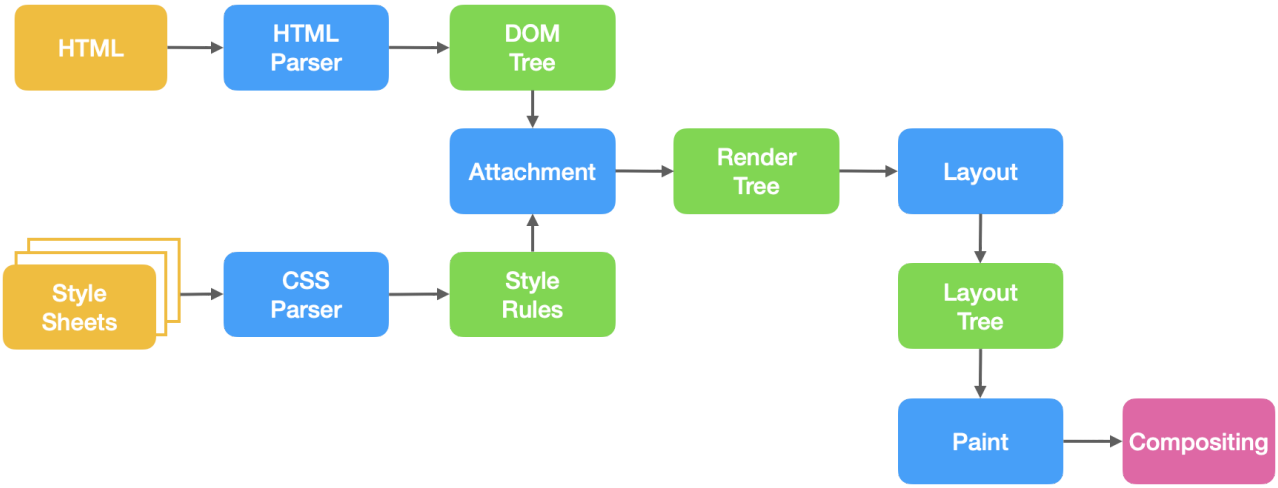
5. 什么是 SSR

5.1 SSR 定义

页面的渲染流程：

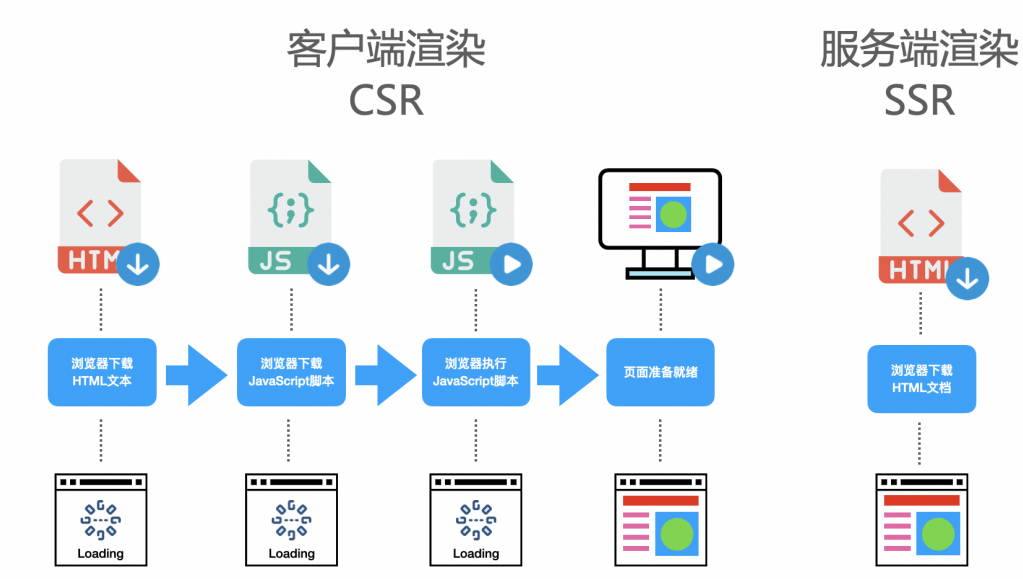
1. 浏览器通过请求得到一个HTML文本；
2. 渲染进程解析HTML文本，构建DOM树；
3. 解析HTML的同时，如果遇到内联样式或者样式脚本，则下载并构建样式规则（stytle rules），若遇到JavaScript脚本，则会下载执行脚本；
4. DOM树和样式规则构建完成之后，渲染进程将两者合并成渲染树（render tree）；

5. 渲染进程开始对渲染树进行布局，生成布局树（layout tree）；
6. 渲染进程对布局树进行绘制，生成绘制记录；
7. 渲染进程的对布局树进行分层，分别栅格化每一层，并得到合成帧；
8. 渲染进程将合成帧信息发送给GPU进程显示到页面中；



可以看到，页面的渲染其实就是浏览器将HTML文本转化为页面帧的过程。而如今我们大部分WEB应用都是使用 JavaScript 框架（Vue、React、Angular）进行页面渲染的，也就是说，在执行 JavaScript 脚本的时候，HTML页面已经开始解析并且构建DOM树了，JavaScript 脚本只是动态的改变 DOM 树的结构，使得页面成为希望成为的样子，这种渲染方式叫动态渲染，也可以叫客户端渲染（client side rende）；

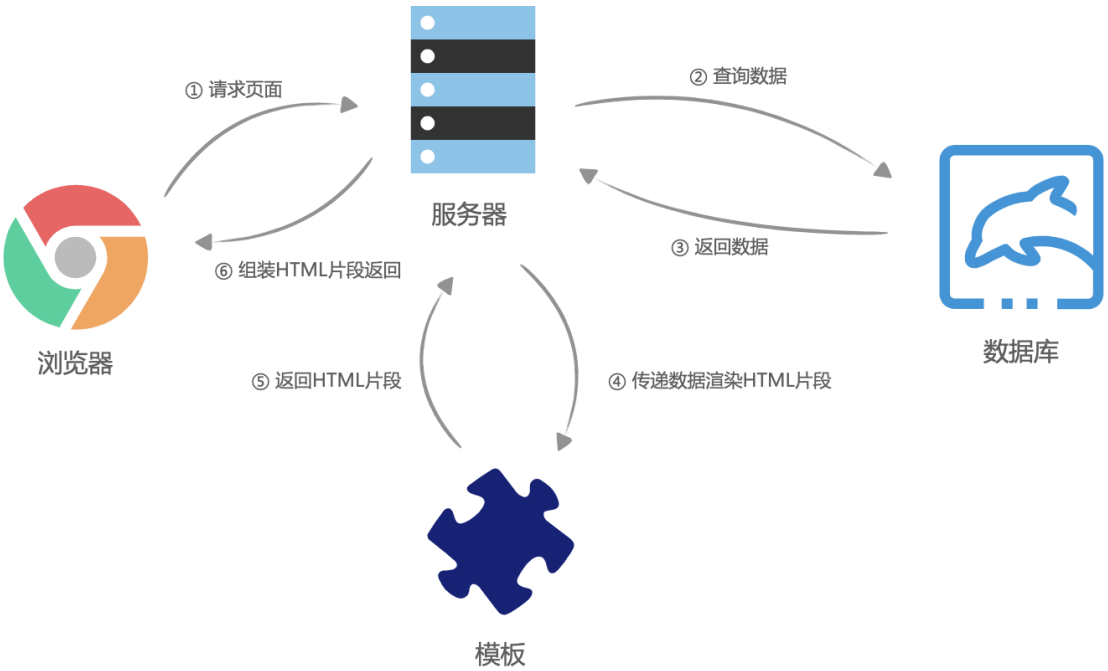
那么什么是服务端渲染（server side render）？顾名思义，服务端渲染就是在浏览器请求页面URL的时候，服务端将我们需要的HTML文本组装好，并返回给浏览器，这个HTML文本被浏览器解析之后，不需要经过 JavaScript 脚本的执行，即可直接构建出希望的 DOM 树并展示到页面中。这个服务端组装HTML的过程，叫做服务端渲染；



5.2 SSR的由来

5.2.1 Web1.0

在没有AJAX的时候，也就是web1.0时代，几乎所有应用都是服务端渲染（此时服务器渲染非现在的服务器渲染），那个时候的页面渲染大概是这样的，浏览器请求页面URL，然后服务器接收到请求之后，到数据库查询数据，将数据丢到后端的组件模板（php、asp、jsp等）中，并渲染成HTML片段，接着服务器在组装这些HTML片段，组成一个完整的HTML，最后返回给浏览器，这个时候，浏览器已经拿到了一个完整的被服务器动态组装出来的HTML文本，然后将HTML渲染到页面中，过程没有任何JavaScript代码的参与。



5.2.2 客户端渲染

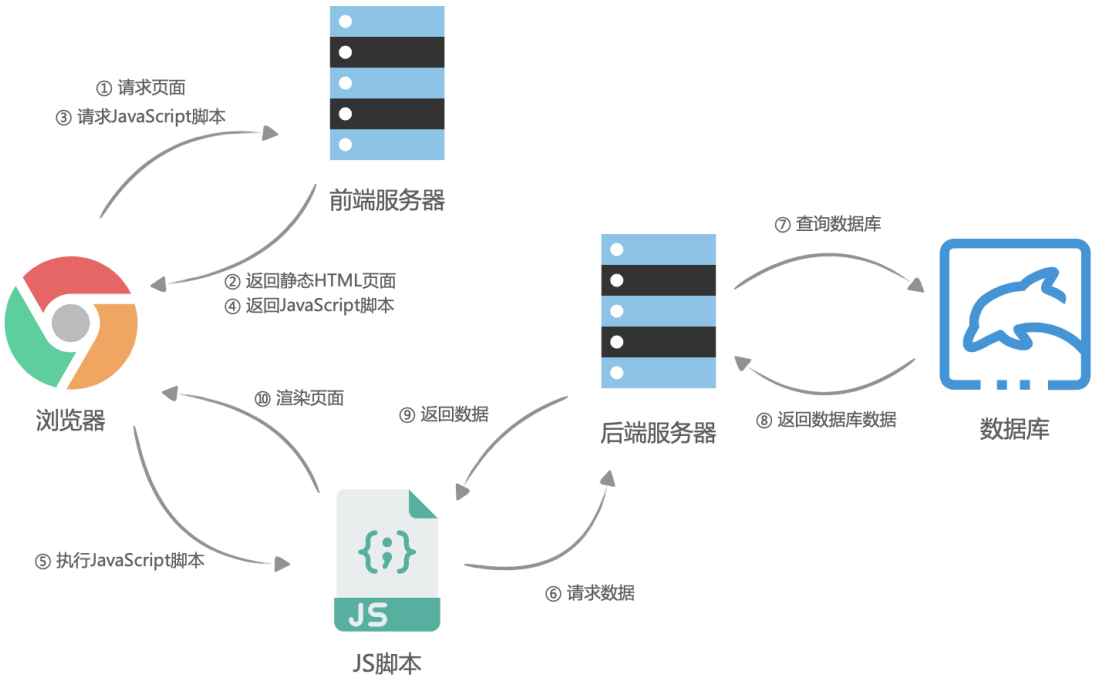
在WEB1.0时代，服务端渲染看起来是一个当时的最好的渲染方式，但是随着业务的日益复杂和后续AJAX的出现，也渐渐开始暴露出了WEB1.0服务器渲染的缺点。

- 每次更新页面的一小的模块，都需要重新请求一次页面，重新查一次数据库，重新组装一次HTML
- 前端JavaScript代码和后端（jsp、php、jsp）代码混杂在一起，使得日益复杂的WEB应用难以维护

而且那个时候，根本就没有前端工程师这一职位，前端js的活一般都由后端同学 jQuery 一把梭。但是随着前端页面渐渐地复杂了之后，后端开始发现js好麻烦，虽然很简单，但是坑太多了，于是让公司招聘了一些专门写js的人，也就是前端，这个时候，前后端的鄙视链就出现了，后端鄙视前端，因为后端觉得js太简单，无非就是写写页面的特效（JS），切切图（CSS），根本算不上是真正的程序员。

随之 nodejs 的出现，前端看到了翻身的契机，为了摆脱后端的指指点点，前端开启了一场前后端分离的运动，希望可以脱离后端独立发展。前后端分离，表面上看上去是代码分离，实际上是为了前后端人员分离，也就是前后端分家，前端不再归属于后端团队。

前后端分离之后，网页开始被当成了独立的应用程序（SPA，Single Page Application），前端团队接管了所有页面渲染的事，后端团队只负责提供所有数据查询与处理的API，大体流程是这样的：首先浏览器请求URL，前端服务器直接返回一个空的静态HTML文件（不需要任何查数据库和模板组装），这个HTML文件中加载了很多渲染页面需要的 JavaScript 脚本和 CSS 样式表，浏览器拿到 HTML 文件后开始加载脚本和样式表，并且执行脚本，这个时候脚本请求后端服务提供的API，获取数据，获取完成后将数据通过JavaScript脚本动态的将数据渲染到页面中，完成页面显示。

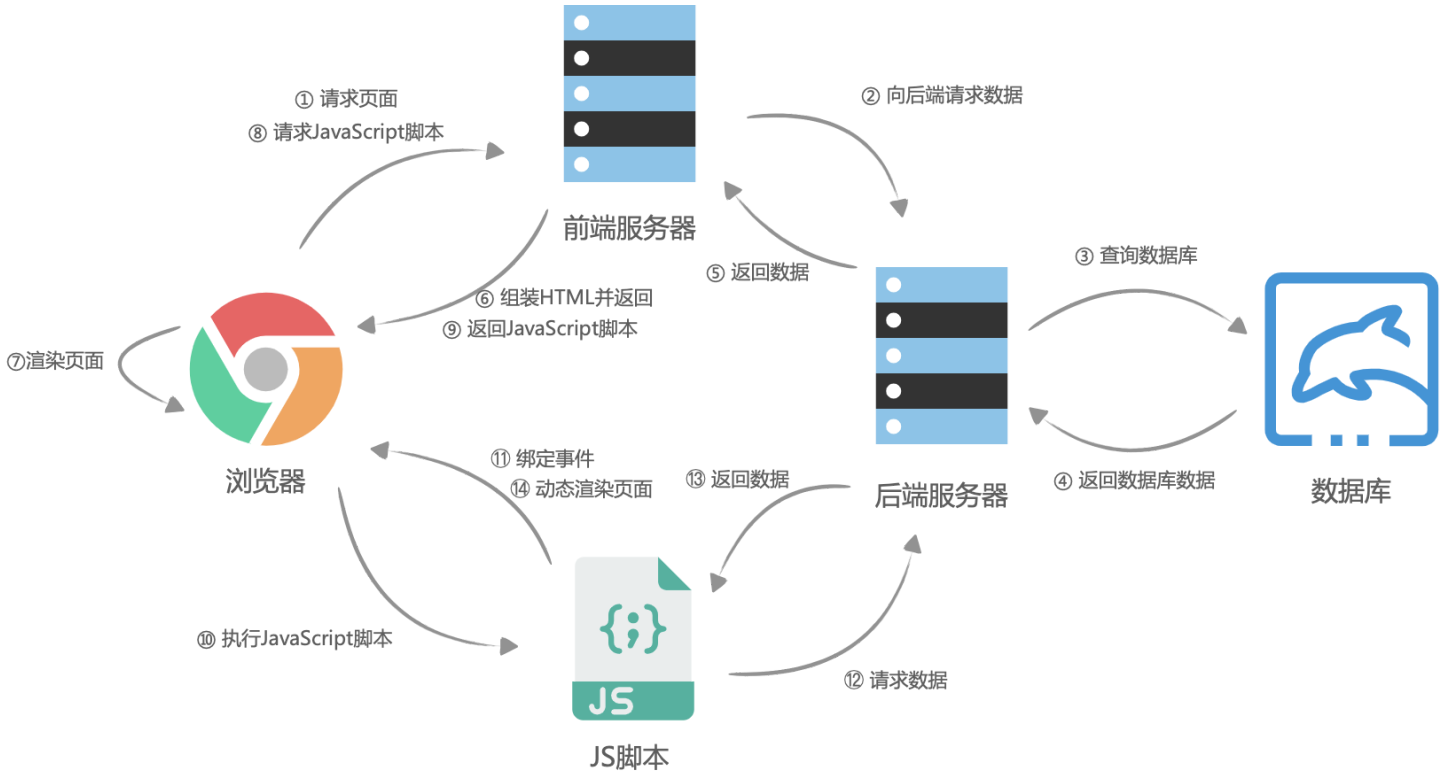


这一个前后端分离的渲染模式，也就是客户端渲染（CSR）。

5.2.3 服务端渲染

随着单页应用（SPA）的发展，程序员们渐渐发现 SEO（Search Engine Optimazition，即搜索引擎优化）出了问题，而且随着应用的复杂化，JavaScript 脚本也不断的臃肿起来，使得首屏渲染相比于Web1.0时候的服务端渲染，也慢了不少。

自己选的路，跪着也要走下去。于是前端团队选择了使用 nodejs 在服务器进行页面的渲染，进而再次出现了服务端渲染。大体流程与客户端渲染有些相似，首先是浏览器请求URL，前端服务器接收到URL请求之后，根据不同的URL，前端服务器向后端服务器请求数据，请求完成后，前端服务器会组装一个携带了具体数据的HTML文本，并且返回给浏览器，浏览器得到HTML之后开始渲染页面，同时，浏览器加载并执行 JavaScript 脚本，给页面上的元素绑定事件，让页面变得可交互，当用户与浏览器页面进行交互，如跳转到下一个页面时，浏览器会执行 JavaScript 脚本，向后端服务器请求数据，获取完数据之后再次执行 JavaScript 代码动态渲染页面。



5.3 服务端渲染的利弊

相比于客户端渲染，服务端渲染有什么优势？

5.3.1 好处

- 利于SEO

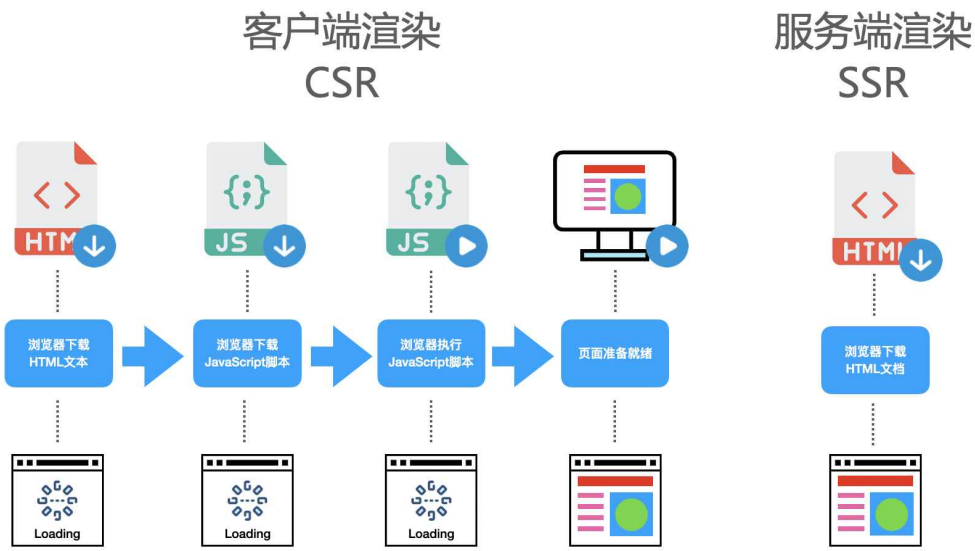
有利于SEO，其实就是有利于爬虫来爬你的页面，然后在别人使用搜索引擎搜索相关的内容时，你的网页排行能靠得更前，这样你的流量就有越高。那为什么服务端渲染更利于爬虫爬你的页面呢？其实，爬虫也分低级爬虫和高级爬虫。

- 低级爬虫：只请求URL，URL返回的HTML是什么内容就爬什么内容。
- 高级爬虫：请求URL，加载并执行JavaScript脚本渲染页面，爬JavaScript渲染后的内容。

也就是说，低级爬虫对客户端渲染的页面来说，简直无能为力，因为返回的HTML是一个空壳，它需要执行 JavaScript 脚本之后才会渲染真正的页面。而目前像百度、谷歌、微软等公司，有一部分年代老旧的爬虫还属于低级爬虫，使用服务端渲染，对这些低级爬虫更加友好一些。

- 白屏时间更短

相对于客户端渲染，服务端渲染在浏览器请求URL之后已经得到了一个带有数据的HTML文本，浏览器只需要解析HTML，直接构建DOM树就可以。而客户端渲染，需要先得到一个空的HTML页面，这个时候页面已经进入白屏，之后还需要经过加载并执行 JavaScript、请求后端服务器获取数据、JavaScript 渲染页面几个过程才可以看到最后的页面。特别是在复杂应用中，由于需要加载 JavaScript 脚本，越是复杂的应用，需要加载的 JavaScript 脚本就越多、越大，这会导致应用的首屏加载时间非常长，进而降低了体验感。



5.3.2 缺点

并不是所有的WEB应用都必须使用SSR，这需要开发者自己来权衡，因为服务端渲染会带来以下问题：

- 代码复杂度增加。为了实现服务端渲染，应用代码中需要兼容服务端和客户端两种运行情况，而一部分依赖的外部扩展库却只能在客户端运行，需要对其进行特殊处理，才能在服务器渲染应用程序中运行。
- 需要更多的服务器负载均衡。由于服务器增加了渲染HTML的需求，使得原本只需要输出静态资源文件的nodejs服务，新增了数据获取的IO和渲染HTML的CPU占用，如果流量突然暴增，有可能导致服务器down机，因此需要使用响应的缓存策略和准备相应的服务器负载。
- 涉及构建设置和部署的更多要求。与可以部署在任何静态文件服务器上的完全静态单页面应用程序 (SPA) 不同，服务器渲染应用程序，需要处于 Node.js server 运行环境。

5.4 Vue SSR

5.4.1 实现一个基础的SSR应用

核心使用 `createSSRApp` 和 `express` 实现：

```
import { createSSRApp } from 'vue';

export function createApp() {
  return createSSRApp({
    data: () => ({ count: 1 }),
    template: `<div @click="count++">{{ count }}</div>`,
  });
}
```

```
import express from 'express';
import { renderToString } from 'vue/server-renderer';
import { createApp } from './app.js';

const server = express();

server.get('/', (req, res) => {
  const app = createApp();

  renderToString(app).then(html => {
    res.send(`
    <!DOCTYPE html>
    <html>
      <head>
        <title>Vue SSR Example</title>
        <script type="importmap">
          {
            "imports": {
              "vue": "https://unpkg.com/vue@3/dist/vue.esm-browser.js"
            }
          }
        </script>
        <script type="module" src="/client.js"></script>
      </head>
      <body>
        <div id="app">${html}</div>
      </body>
    </html>
    `);
  });
});

server.use(express.static('.'));

server.listen(3000, () => {
  console.log('ready');
});
```

5.4.2 常见方案

Nuxt

Nuxt 是一个构建于 Vue 生态系统之上的全栈框架，它为编写 Vue SSR 应用提供了丝滑的开发体验。更棒的是，你还可以把它当作一个静态站点生成器来用！我们强烈建议你试一试。

Quasar

Quasar 是一个基于 Vue 的完整解决方案，它可以让你用同一套代码库构建不同目标的应用，如 SPA、SSR、PWA、移动端应用、桌面端应用以及浏览器插件。除此之外，它还提供了一整套 Material Design 风格的组件库。

Vite SSR

Vite 提供了内置的 Vue 服务端渲染支持，但它在设计上是偏底层的。如果你想要直接使用 Vite，可以看看 vite-plugin-ssr，一个帮你抽象掉许多复杂细节的社区插件。