服务端渲染SSR

目标

- ssr概念
- vue ssr原生实现
- nuxt.js

资源

1. vue ssr
2. nuxt.js

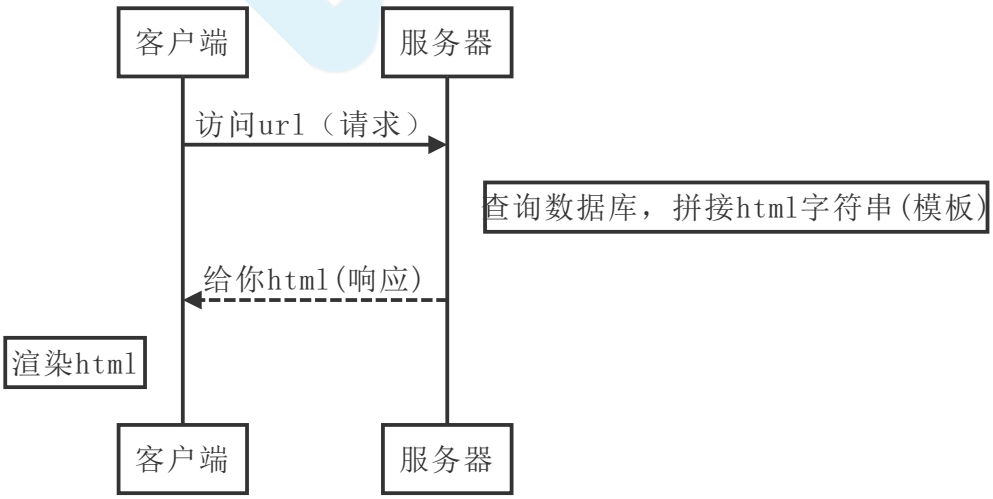
知识点

理解ssr

传统服务端渲染SSR VS 单页面应用SPA VS 服务端渲染SSR

传统web开发

传统web开发，网页内容在服务端渲染完成，一次性传输到浏览器。



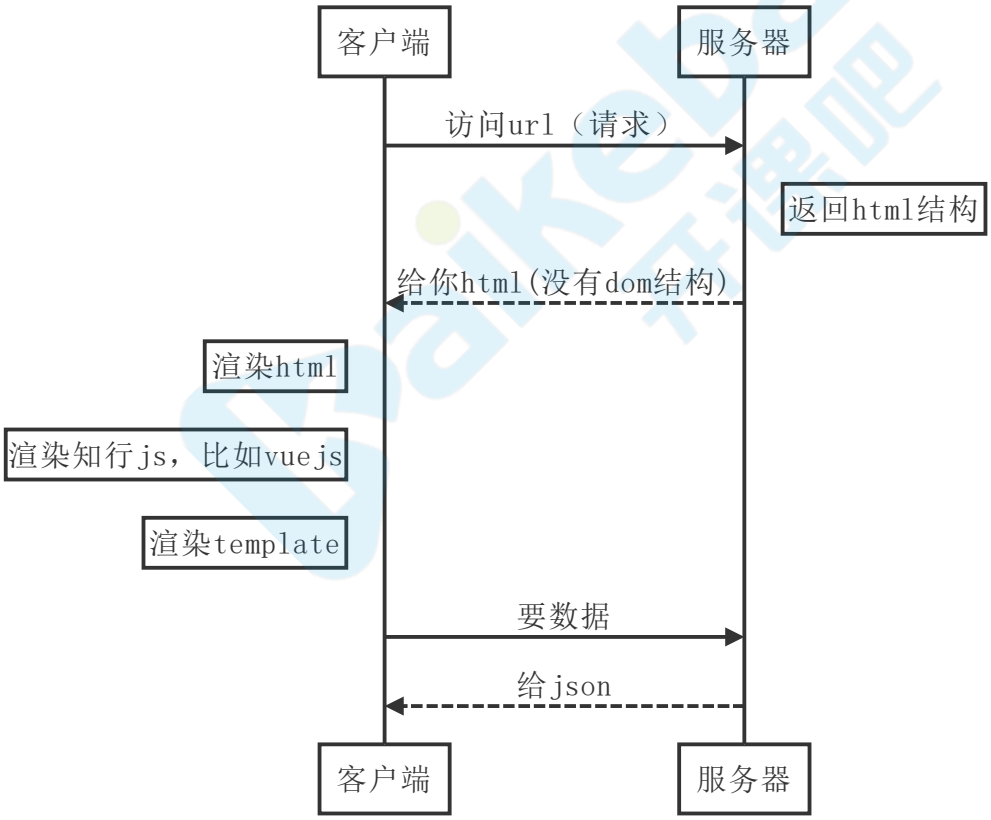
测试代码，server\01-express-test.js

打开页面查看源码，浏览器拿到的是全部的dom结构

```
        <html>
            <div>
                <div id="app">
                    <h1>开课吧</h1>
                    <p class="demo">开课吧还不错</p>
                </div>
            </body>
        </html>
```

单页应用 Single Page App

单页应用优秀的用户体验，使其逐渐成为主流，页面内容由JS渲染出来，这种方式称为客户端渲染。



测试：`npm run serve`

打开页面查看源码，浏览器拿到的仅有宿主元素#app，并没有内容。

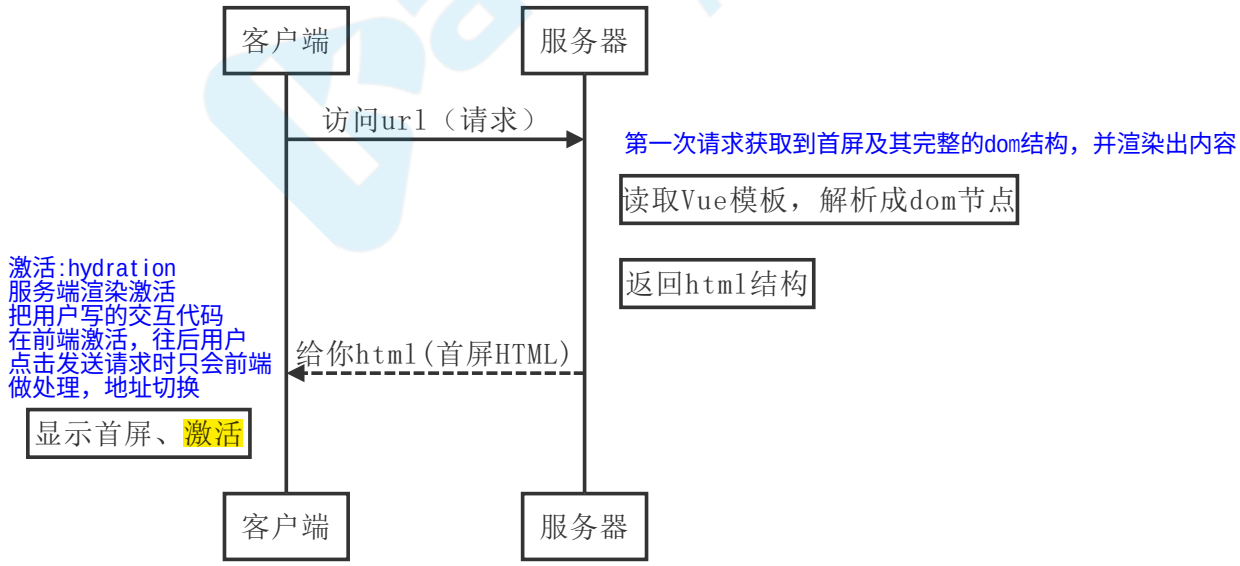
```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width,initial-scale=1
    <link rel="icon" href="/favicon.ico">
    <title>study-vue</title>
  <link href="/js/app.js" rel="preload" as="script"><link href="/js/cl
  <body>
    <noscript>
      <strong>We're sorry but study-vue doesn't work properly without
    </noscript>
    <div id="app"></div>
    <!-- built files will be auto injected -->
  <script type="text/javascript" src="/js/chunk-vendors.js"></script>
(Alt + A)ml>
```

spa缺点:

- seo
- 首屏内容到达时间

服务端渲染 Server Side Render

SSR解决方案，后端渲染出完整的首屏的dom结构返回，前端拿到的内容包括**首屏及完整spa结构**，应用激活后依然按照spa方式运行，这种页面渲染方式被称为服务端渲染 (server side render)



Vue SSR实战

新建工程

vue-cli创建工程即可

```
vue create ssr
```

> 演示项目使用vue-cli 4.x创建

安装依赖

```
npm install vue vue-server-renderer express -D
```

> 确保vue、vue-server-renderer、vue-template-compiler版本一致

启动脚本

创建一个express服务器，将vue ssr集成进来，./server/02-simple-ssr.js

```
// 导入express作为渲染服务器
const express = require("express");
// 导入Vue用于声明待渲染实例
const Vue = require("vue");
// 导入createRenderer用于获取渲染器
const { createRenderer } = require("vue-server-renderer");
// 创建express实例
const app = express();
// 获取渲染器
const renderer = createRenderer();
每次渲染创建一个新的vue实例
// 待渲染vue实例
const vm = new Vue({
  data: {
    name: "开课吧"
  },
  template: `
    <div >
        <h1>{{name}}</h1>
    </div>
  `
});

app.get("/", async function(req, res) {
  // renderToString可以将vue实例转换为html字符串
  // 若未传递回调函数，则返回Promise
  try {
    const html = await renderer.renderToString(vm);
    res.send(html);
  } catch (error) {
    res.status(500).send("Internal Server Error");
  }
});

app.listen(3000, () => {
  // eslint-disable-next-line no-console
  console.log("启动成功");
});
```

路由

路由支持仍然使用vue-router

安装

若未引入vue-router则需要安装

```
npm i vue-router -s
```

```
vue add router
```

配置

创建@/router/index.js

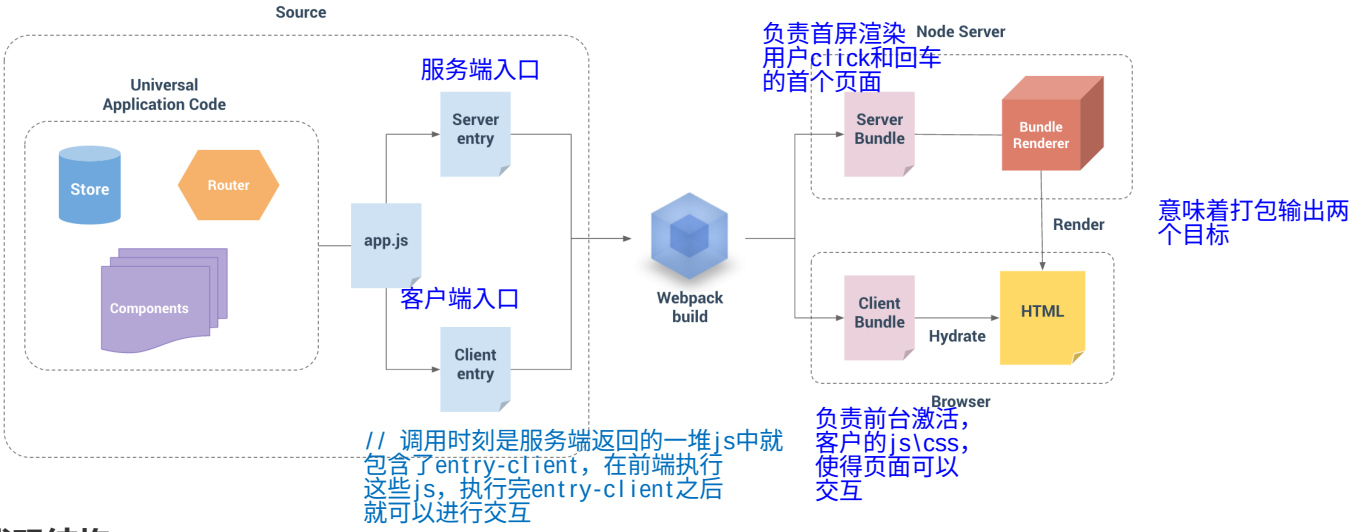
```
import Vue from "vue";
import Router from "vue-router";
import Home from "@/views/Home";
import About from "@/views/About";

Vue.use(Router);

//导出工厂函数
export function createRouter() {
  return new Router({
    routes: [
      { path: "/", component: Home },
      { path: "/about", component: About }
    ]
  });
}
```

构建

对于客户端应用程序和服务器应用程序，我们都要使用 webpack 打包 - 服务器需要「服务器 bundle」然后用于服务器端渲染(SSR)，而「客户端 bundle」会发送给浏览器，用于混合静态标记。

构建流程

*实际上进行请求的只有首屏页面，其他只是客户端路由切换



代码结构

```
src
├── main.js # 用于创建vue实例
├── entry-client.js # 客户端入口，用于静态内容“激活”
└── entry-server.js # 服务端入口，用于首屏内容渲染
```

Vue实例创建

`main.js` 是负责创建vue实例，每次请求均会有独立的vue实例创建。创建main.js：

```
import Vue from "vue";
import App from "./App.vue";
import { createRouter } from "./router";

// 导出Vue实例工厂函数，为每次请求创建独立实例
// 上下文用于给vue实例传递参数
export function createApp(context) {
  const router = createRouter();
  const app = new Vue({
    router,
    context,
    render: h => h(App)
  });
  return { app, router };
}
```

服务端入口

服务端入口文件src/entry-server.js

```
import { createApp } from "./main";

// 返回一个函数，接收请求上下文，返回创建的vue实例
export default context => {
  // 这里返回一个Promise，确保路由或组件准备就绪
```

```
  return new Promise((resolve, reject) => {
    const { app, router } = createApp(context);
    // 跳转到首屏的地址
    router.push(context.url);
    // 路由就绪，返回结果
    router.onReady(() => {
      resolve(app);
    }, reject);
  });
};
```

客户端入口

客户端入口只需创建vue实例并执行挂载，这一步称为激活。创建entry-client.js：

```
import { createApp } from "./main";

// 创建vue、router实例
const { app, router } = createApp();
// 路由就绪，执行挂载
router.onReady(() => {
  app.$mount("#app");
});
```

webpack配置

安装依赖

```
npm install webpack-node-externals lodash.merge -D
```

具体配置，vue.config.js

```
// 两个插件分别负责打包客户端和服务端
const VueSSRServerPlugin = require("vue-server-renderer/server-plugin");
const VueSSRClientPlugin = require("vue-server-renderer/client-plugin");
const nodeExternals = require("webpack-node-externals");
const merge = require("lodash.merge");
// 根据传入环境变量决定入口文件和相应配置项
const TARGET_NODE = process.env.WEBPACK_TARGET === "node";
const target = TARGET_NODE ? "server" : "client";

module.exports = {
  css: {
    extract: false
  },
  outputDir: './dist/'+target,
  configureWebpack: () => ({
    // 将 entry 指向应用程序的 server / client 文件
    entry: `./src/entry-${target}.js`,
    // 对 bundle renderer 提供 source map 支持
    devtool: 'source-map',
    // target设置为node使webpack以Node适用的方式处理动态导入，
```

```
    // 并且还会在编译Vue组件时告知`vue-loader`输出面向服务器代码。
    target: TARGET_NODE ? "node" : "web",
    // 是否模拟node全局变量
    node: TARGET_NODE ? undefined : false,
    output: {
      // 此处使用Node风格导出模块
      libraryTarget: TARGET_NODE ? "commonjs2" : undefined
    },
    // https://webpack.js.org/configuration/externals/#function
    // https://github.com/liady/webpack-node-externals
    // 外置化应用程序依赖模块。可以使服务器构建速度更快，并生成较小的打包文件。
    externals: TARGET_NODE
      ? nodeExternals({
          // 不要外置化webpack需要处理的依赖模块。
          // 可以在这里添加更多的文件类型。例如，未处理 *.vue 原始文件，
          // 还应该将修改`global`（例如polyfill）的依赖模块列入白名单
          whitelist: [/\.css$/]
        })
      : undefined,
    optimization: {
      splitChunks: undefined
    },
    // 这是将服务器的整个输出构建为单个 JSON 文件的插件。
    // 服务端默认文件名为 `vue-ssr-server-bundle.json`
    // 客户端默认文件名为 `vue-ssr-client-manifest.json`。
    plugins: [TARGET_NODE ? new VueSSRServerPlugin() : new VueSSRClientPlugin()]
  }),
  chainWebpack: config => {
    // cli4项目添加
    if (TARGET_NODE) {
        config.optimization.delete('splitChunks')
    }

    config.module
      .rule("vue")
      .use("vue-loader")
      .tap(options => {
        merge(options, {
          optimizeSSR: false
        });
      });
  }
};
```

> 对应配置文档

脚本配置

安装依赖

```
npm i cross-env -D
```

定义创建脚本，package.json

```
"scripts": {
    "build:client": "vue-cli-service build",
    "build:server": "cross-env WEBPACK_TARGET=node vue-cli-service build",
    "build": "npm run build:server && npm run build:client"
},
```

> 执行打包：npm run build

宿主文件

最后需要定义宿主文件，修改./public/index.html

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width,initial-scale=1.0">
    <title>Document</title>
  </head>
  <body>
    <!--vue-ssr-outlet-->
  </body>
</html>
```

服务器启动文件

修改服务器启动文件，现在需要处理所有路由，./server/04-ssr.js

```
// 加载本地文件
const fs = require("fs");
// 处理url
const path = require("path");
const express = require('express')
const server = express()

// 获取绝对路径
const resolve = dir => {
    return path.resolve(__dirname, dir)
}

// 第 1 步：开放dist/client目录，关闭默认下载index页的选项，不然到不了后面路由
// /index.html
server.use(express.static(resolve('../dist/client'), {index: false}))

// 第 2 步：获得一个createBundleRenderer
const { createBundleRenderer } = require("vue-server-renderer");

// 第 3 步：导入服务端打包文件
const bundle = require(resolve("../dist/server/vue-ssr-server-bundle.json"));

// 第 4 步：创建渲染器
const template = fs.readFileSync(resolve("../public/index.html"), "utf-8");
```

```
const clientManifest = require(resolve("../dist/client/vue-ssr-client-
manifest.json"));
const renderer = createBundleRenderer(bundle, {
  runInNewContext: false, // https://ssr.vuejs.org/zh/api/#runinnewcontext
  template, // 宿主文件
  clientManifest // 客户端清单
});

// 路由是通配符，表示所有url都接受
server.get('*', async (req,res)=>{
    console.log(req.url);

  // 设置url和title两个重要参数
  const context = {
    title:'ssr test',
    url:req.url // 首屏地址
  }
  const html = await renderer.renderToString(context);
  res.send(html)
})

server.listen(3000, function() {
  // eslint-disable-next-line no-console
  console.log(`server started at localhost:${port}`);
});
```

整合Vuex

安装vuex （ssr时也在服务端进行）

```
npm install -S vuex
```

store/index.js

```
import Vue from 'vue'
import Vuex from 'vuex'

Vue.use(Vuex)

export function createStore () {
  return new Vuex.Store({
    state: {
        count:108
    },
    mutations: {
        add(state){
            state.count += 1;
        }
    }
  })
}
```

挂载store，main.js

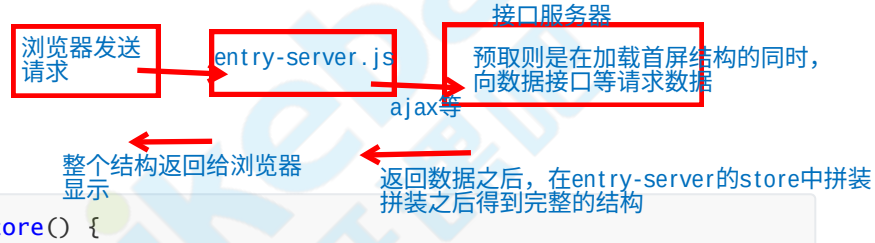
```
import { createStore } from './store'

export function createApp (context) {
    // 创建实例
    const store = createStore()
    const app = new Vue({
        store, // 挂载
        render: h => h(App)
    })
    return { app, router, store }
}
```

使用，.src/components/Index.vue

```
<h2 @click="$store.commit('add')">{{$store.state.count}}</h2>
```

数据预取

服务器端渲染的是应用程序的"快照"，如果应用依赖于一些异步数据，**那么在开始渲染之前，需要先预取和解析好这些数据。**



异步数据获取，store/index.js

```
export function createStore() {
  return new Vuex.Store({
    mutations: {
      // 加一个初始化
      init(state, count) {
        state.count = count;
      },
    },
    actions: {
      // 加一个异步请求count的action
      getCount({ commit }) {
        return new Promise(resolve => {
          setTimeout(() => {
            commit("init", Math.random() * 100);
            resolve();
          }, 1000);
        });
      },
    },
  });
}
```

组件中的数据预取逻辑，Index.vue

```
export default {
  asyncData({ store, route }) { // 约定预取逻辑编写在预取钩子asyncData中
    // 触发 action 后，返回 Promise 以便确定请求结果
    return store.dispatch("getCount");
  }
};
```

服务端数据预取，entry-server.js

```
import { createApp } from "./app";

export default context => {
  return new Promise((resolve, reject) => {
    // 拿出store和router实例
    const { app, router, store } = createApp(context);
    router.push(context.url);
    router.onReady(() => {
      // 获取匹配的路由组件数组
      const matchedComponents = router.getMatchedComponents();

      // 若无匹配则抛出异常
      if (!matchedComponents.length) {
        return reject({ code: 404 });
      }

      // 对所有匹配的路由组件调用可能存在的`asyncData()`
      Promise.all(
        matchedComponents.map(Component => {
          if (Component.asyncData) {
            return Component.asyncData({
              store,
              route: router.currentRoute,
            });
          }
        }),
      )
        .then(() => {
          // 所有预取钩子 resolve 后，
          // store 已经填充入渲染应用所需状态
          // 将状态附加到上下文，且 `template` 选项用于 renderer 时，
          // 状态将自动序列化为 `window.__INITIAL_STATE__`，并注入 HTML。
          context.state = store.state;

          resolve(app);
        })
        .catch(reject);
    }, reject);
  });
};
```

客户端在挂载到应用程序之前，store 就应该获取到状态，entry-client.js

```
// 导出store
const { app, router, store } = createApp();

// 当使用 template 时，context.state 将作为 window.__INITIAL_STATE__ 状态自动嵌入到最终的 HTML // 在客户端挂载到应用程序之前，store 就应该获取到状态：
if (window.__INITIAL_STATE__) {
  store.replaceState(window.__INITIAL_STATE__);
}
```

> 首屏没有问题，其他页面出现状态异常，客户端没有执行异步代码。

客户端数据预取处理，main.js

```
Vue.mixin({
  beforeMount() {
    const { asyncData } = this.$options;
    if (asyncData) {
      // 将获取数据操作分配给 promise
      // 以便在组件中，我们可以在数据准备就绪后
      // 通过运行 `this.dataPromise.then(...)` 来执行其他任务
      this.dataPromise = asyncData({
        store: this.$store,
        route: this.$route,
      });
    }
  },
});
```

总结

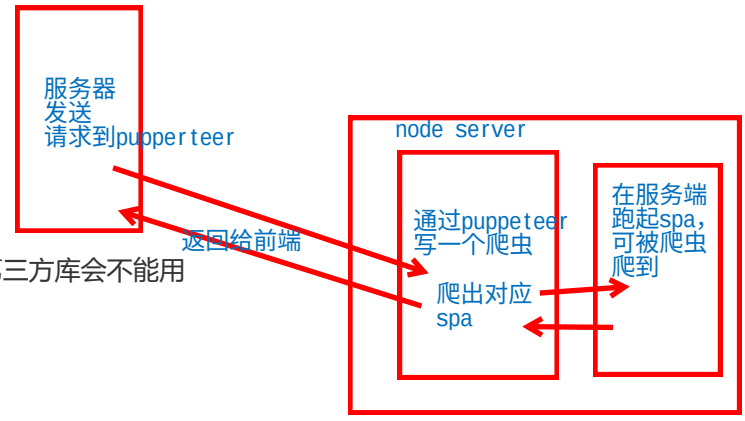
SSR优缺点都很明显

优点：

- seo
- 首屏显示时间

缺点：

- 开发逻辑复杂
- 开发条件限制：比如一些生命周期不能用，一些第三方库会不能用
- 服务器负载大



已经存在spa

- 需要seo页面是否只是少数几个营销页面预渲染是否可以考虑
- 确实需要做ssr改造，利用服务器端爬虫技术puppeteer
- 最后选择重构

Nuxt.js实战

Nuxt.js 是一个基于 Vue.js 的**通用应用框架。**

通过对客户端/服务端基础架构的抽象组织，Nuxt.js 主要关注的是应用的 **UI渲染。**

资源

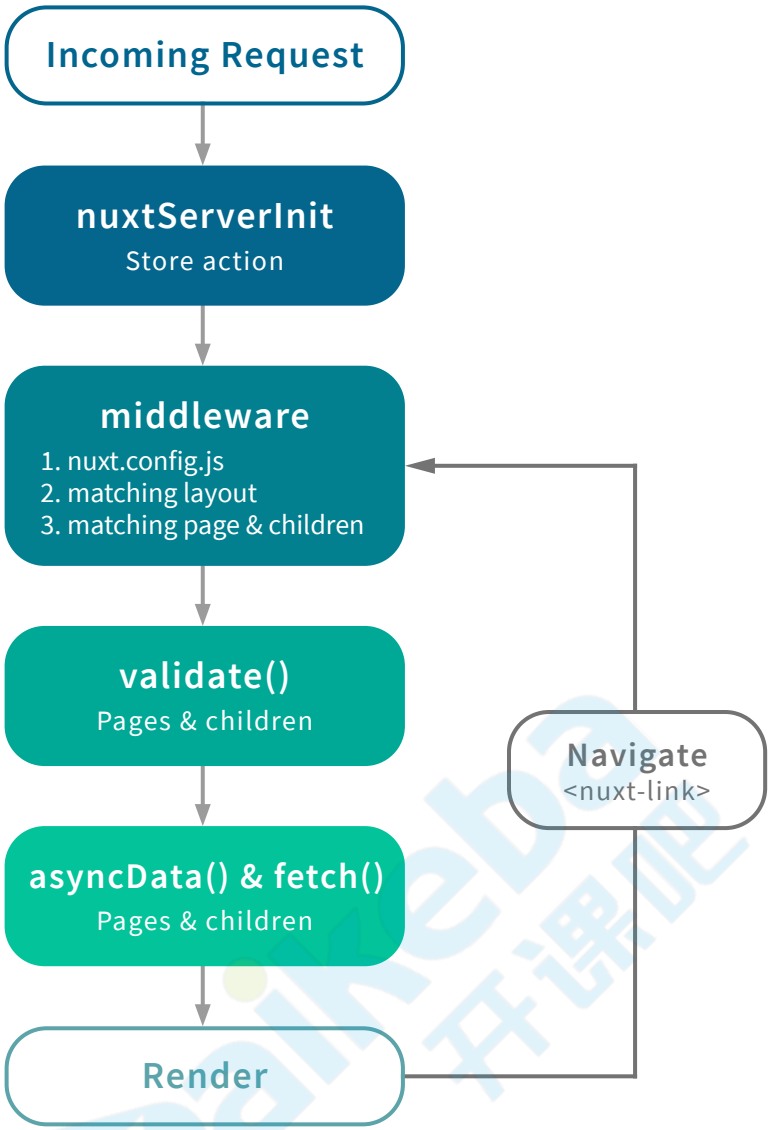
Nuxt.js官方文档

nuxt.js特性

- 代码分层
- 服务端渲染
- 强大的路由功能
- 静态文件服务
- ...

nuxt渲染流程

一个完整的服务器请求到渲染的流程



nuxt安装

运行 create-nuxt-app

```
npx create-nuxt-app <项目名>
```

选项

```
PS C:\Users\yt037\Desktop\kaikeba\projects> npx create-nuxt-app nuxt-app
npx: 341 安装成功，用时 27.05 秒

create-nuxt-app v2.10.1
✨ Generating Nuxt.js project in nuxt-app
? Project name nuxt-app
? Project description My terrific Nuxt.js project
? Author name yt0379
? Choose the package manager Npm
? Choose UI framework None
? Choose custom server framework Koa
? Choose Nuxt.js modules Axios
? Choose linting tools (Press <space> to select, <a> to toggle all, <i> to invert selection)
? Choose test framework None
? Choose rendering mode Universal (SSR)
? Choose development tools jsconfig.json (Recommended for VS Code)
- Installing packages with npm
```

运行项目：`npm run dev`

案例

实现如下功能点

- 服务端渲染
- 权限控制
- 全局状态管理
- 数据接口调用

路由

路由生成

pages目录中所有 `*.vue` 文件自动生成应用的路由配置，新建：

- pages/admin.vue 商品管理页
- pages/login.vue 登录页

> 访问http://localhost:3000/试试，并查看.nuxt/router.js验证生成路由

导航

添加路由导航，layouts/default.vue

```
<nav>
    <nuxt-link to="/">首页</nuxt-link>
    <!--别名：n-link，NLink，NuxtLink-->
    <NLink to="/admin">管理</NLink>
    <n-link to="/cart">购物车</n-link>
</nav>
```

> 功能和router-link等效
>
> 禁用预加载行为：`<n-link no-prefetch>page not pre-fetched</n-link>`

商品列表，index.vue

```
<template>
  <div>
    <h2>商品列表</h2>
    <ul>
      <li v-for="good in goods" :key="good.id" >
        <nuxt-link :to="`/detail/${good.id}`">
          <span>{{good.text}}</span>
          <span>￥{{good.price}}</span>
        </nuxt-link>
      </li>
    </ul>
  </div>
</template>
```

```
<script>
export default {
  data() {
    return { goods: [
      {id:1, text:'Web全栈架构师',price:8999},
      {id:2, text:'Python全栈架构师',price:8999},
    ] }
  }
};
</script>
```

动态路由

以下划线作为前缀的 .vue文件 或 目录会被定义为动态路由，如下面文件结构

```
pages/
--| detail/
----| _id.vue
```

会生成如下路由配置：

```
{
    path: "/detail/:id?",
    component: _9c9d895e,
    name: "detail-id"
}
```

> 如果detail/里面不存在index.vue，:id将被作为可选参数

嵌套路由

创建内嵌子路由，你需要添加一个 .vue 文件，同时添加一个**与该文件同名**的目录用来存放子视图组件。

构造文件结构如下：

```
pages/
--| detail/
----| _id.vue
--| detail.vue
```

生成的路由配置如下：

```
{
  path: '/detail',
  component: 'pages/detail.vue',
  children: [
    {path: ':id?', name: "detail-id"}
  ]
}
```

测试代码，detail.vue

```
<template>
    <div>
        <h2>detail</h2>
        <nuxt-child></nuxt-child>
    </div>
</template>
```

> nuxt-child等效于router-view

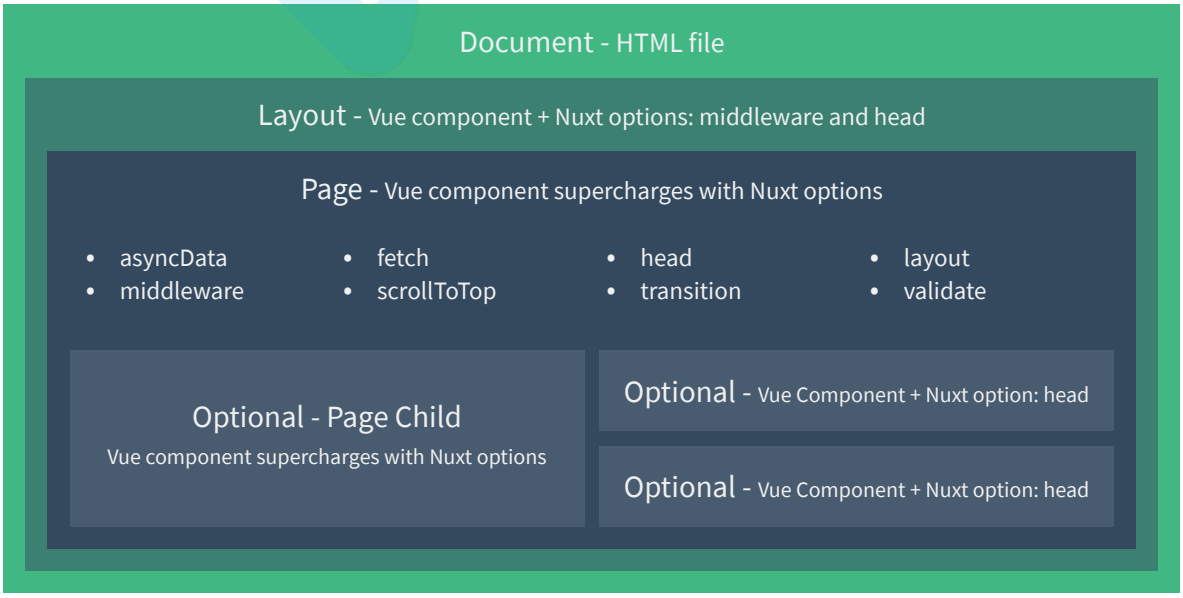
配置路由

要扩展`Nuxt.js`创建的路由，可以通过`router.extendRoutes`选项配置。例如添加自定义路由:

```
// nuxt.config.js
export default {
  router: {
    extendRoutes (routes, resolve) {
      routes.push({
        name: "foo",
        path: "/foo",
        component: resolve(__dirname, "pages/custom.vue")
      });
    }
  }
}
```

视图

下图展示了Nuxt.js 如何为指定的路由配置数据和视图



默认布局

查看`layouts/default.vue`

```
<template>
  <nuxt/>
</template>
```

自定义布局

创建空白布局页面`layouts/blank.vue`，用于login.vue

```
<template>
  <div>
    <nuxt />
  </div>
</template>
```

页面`pages/login.vue`使用自定义布局：

```
export default {
    layout: 'blank'
}
```

自定义错误页面

创建layouts/error.vue

```
<template>
  <div class="container">
    <h1 v-if="error.statusCode === 404">页面不存在</h1>
    <h1 v-else>应用发生错误异常</h1>
    <p>{{error}}</p>
    <nuxt-link to="/">首 页</nuxt-link>
  </div>
</template>

<script>
export default {
  props: ['error'],
  layout:'blank'
}
</script>
```

测试：访问一个不存在的页面

页面

页面组件就是 Vue 组件，只不过 Nuxt.js 为这些组件添加了一些特殊的配置项

给首页添加标题和meta等，index.vue

```
export default {
  head() {
    return {
      title: "课程列表",
      // vue-meta利用hid确定要更新meta
      meta: [{ name: "description", hid: "description", content: "set page meta"
}],
      link: [{ rel: "favicon", href: "favicon.ico" }],
    };
  },
};
```

> 更多页面配置项

异步数据获取

`asyncData` 方法使得我们可以在**设置组件数据之前异步获取或处理数据。**

范例：获取商品数据

接口准备

- 安装依赖：`npm i koa-router koa-bodyparser -S`
- 接口文件，server/api.js

整合axios

安装@nuxt/axios模块：`npm install @nuxtjs/axios -S`

配置：nuxt.config.js

```
modules: [
    '@nuxtjs/axios',
],
axios: {
    proxy: true
},
proxy: {
    "/api": "http://localhost:8080"
},
```

> 注意配置重启生效

测试代码：获取商品列表，index.vue

```
<script>
export default {
    async asyncData({ $axios, error }) {
        const {ok, goods} = await $axios.$get("/api/goods");
        if (ok) {
            return { goods };
        }
        // 错误处理
        error({ statusCode: 400, message: "数据查询失败" });
    },
}
</script>
```

测试代码：获取商品详情，/index/_id.vue

```
<template>
  <div>
    <pre v-if="goodInfo">{{goodInfo}}</pre>
  </div>
</template>

<script>
export default {
  async asyncData({ $axios, params, error }) {
    if (params.id) {
      // asyncData中不能使用this获取组件实例
      // 但是可以通过上下文获取相关数据
      const { data: goodInfo } = await $axios.$get("/api/detail", { params });

      if (goodInfo) {
        return { goodInfo };
      }
      error({ statusCode: 400, message: "商品详情查询失败" });
    } else {
      return { goodInfo: null };
    }
  }
};
</script>
```

中间件

中间件会在一个页面或一组页面渲染之前运行我们定义的函数，常用于权限控制、校验等任务。

范例代码：管理员页面保护，创建middleware/auth.js

```
export default function({ route, redirect, store }) {
  // 上下文中通过store访问vuex中的全局状态
  // 通过vuex中令牌存在与否判断是否登录
  if (!store.state.user.token) {
    redirect("/login?redirect="+route.path);
  }
}
```

注册中间件，admin.vue

```
<script>
    export default {
        middleware: ['auth']
    }
</script>
```

> 全局注册：将会对所有页面起作用，nuxt.config.js
>
> ```
> router: {
>     middleware: ['auth']
> },
> ```

> 运行报错，因为不存在user模块

状态管理 vuex

应用根目录下如果存在 `store` 目录，Nuxt.js将启用vuex状态树。定义各状态树时具名导出state, mutations, getters, actions即可。

范例：用户登录及登录状态保存，创建store/user.js

```
export const state = () => ({
  token: ''
});

export const mutations = {
  init(state, token) {
    state.token = token;
  }
};

export const getters = {
  isLogin(state) {
    return !!state.token;
  }
};

export const actions = {
  login({ commit, getters }, u) {
    return this.$axios.$post("/api/login", u).then(({ token }) => {
      if (token) {
        commit("init", token);
```

```
      }
      return getters.isLogin;
    });
  }
};
```

登录页面逻辑，login.vue

```
<template>
  <div>
    <h2>用户登录</h2>
    <el-input v-model="user.username"></el-input>
    <el-input type="password" v-model="user.password"></el-input>
    <el-button @click="onLogin">登录</el-button>
  </div>
</template>

<script>
export default {
  data() {
    return {
      user: {
        username: "",
        password: ""
      }
    };
  },
  methods: {
    onLogin() {
      this.$store.dispatch("user/login", this.user).then(ok=>{
          if (ok) {
            const redirect = this.$route.query.redirect || '/'
            this.$router.push(redirect);
          }
      });
    }
  }
};
</script>
```

插件

Nuxt.js会在运行应用之前执行插件函数，需要引入或设置Vue插件、自定义模块和第三方模块时特别有用。

范例代码：接口注入，利用插件机制将服务接口注入组件实例、store实例中，创建plugins/api-inject.js

```
export default ({ $axios }, inject) => {
  inject("login", user => {
    return $axios.$post("/api/login", user);
  });
};
```

注册插件，nuxt.config.js

```
plugins: [
    "@/plugins/api-inject"
],
```

范例：添加请求拦截器附加token，创建plugins/interceptor.js

```
export default function({ $axios, store }) {
  $axios.onRequest(config => {
    if (store.state.user.token) {
      config.headers.Authorization = "Bearer " + store.state.user.token;
    }
    return config;
  });
}
```

注册插件，nuxt.config.js

```
plugins: ["@/plugins/interceptor"]
```

nuxtServerInit

通过在store的根模块中定义 `nuxtServerInit` 方法，Nuxt.js 调用它的时候会将页面的上下文对象作为第2个参数传给它。当我们想将服务端的一些数据传到客户端时，这个方法非常好用。

范例：登录状态初始化，store/index.js

```
export const actions = {
  nuxtServerInit({ commit }, { app }) {
    const token = app.$cookies.get("token");
    if (token) {
      console.log("nuxtServerInit: token:"+token);
      commit("user/init", token);
    }
  }
};
```

- 安装依赖模块：cookie-universal-nuxt

```
npm i -S cookie-universal-nuxt
```

> 注册, nuxt.config.js
>
> ```
> modules: ["cookie-universal-nuxt"],
> ```

- nuxtServerInit只能写在store/index.js
- nuxtServerInit仅在服务端执行

发布部署

服务端渲染应用部署

先进行编译构建，然后再启动 Nuxt 服务

```
npm run build
npm start
```

> 生成内容在.nuxt/dist中

静态应用部署

Nuxt.js 可依据路由配置将应用静态化，使得我们可以将应用部署至任何一个静态站点主机服务商。

```
npm run generate
```

> 注意渲染和接口服务器都需要处于启动状态
>
> 生成内容再dist中