nuxt + vue3

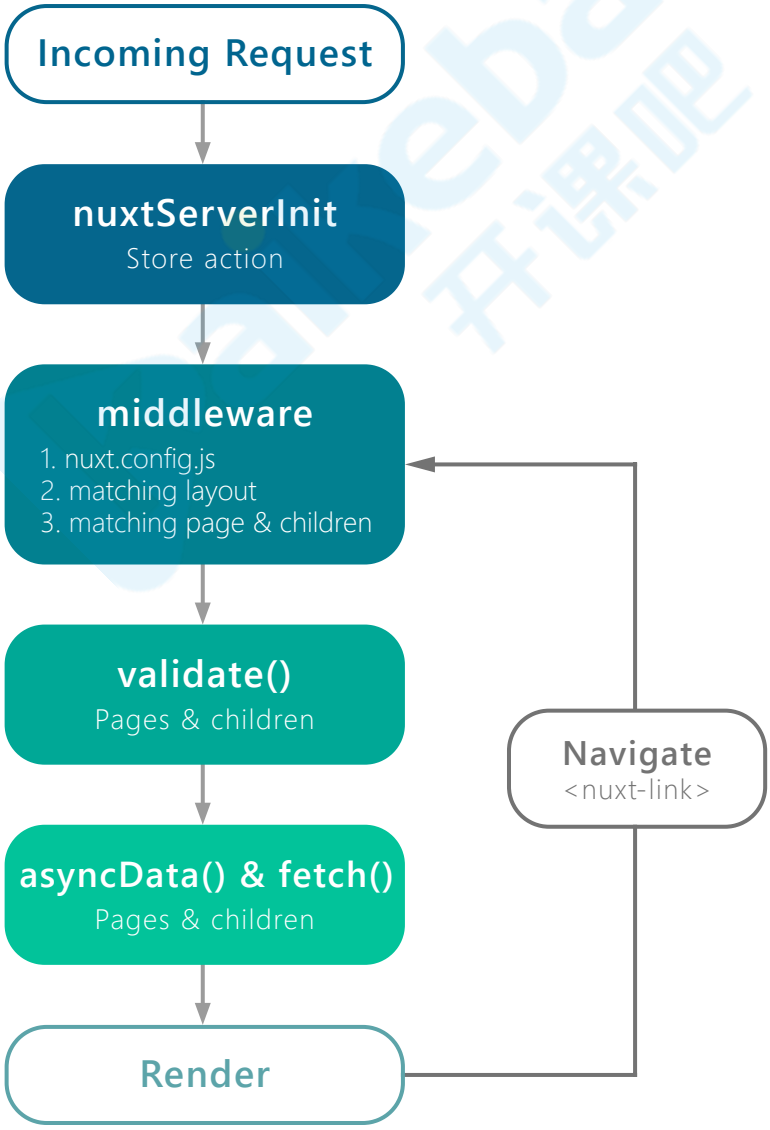
Nuxt.js实战

Nuxt.js 是一个基于 Vue.js 的**通用应用框架。**

资源

Nuxt.js官方文档

nuxt渲染流程



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`

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

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>
```

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>
```

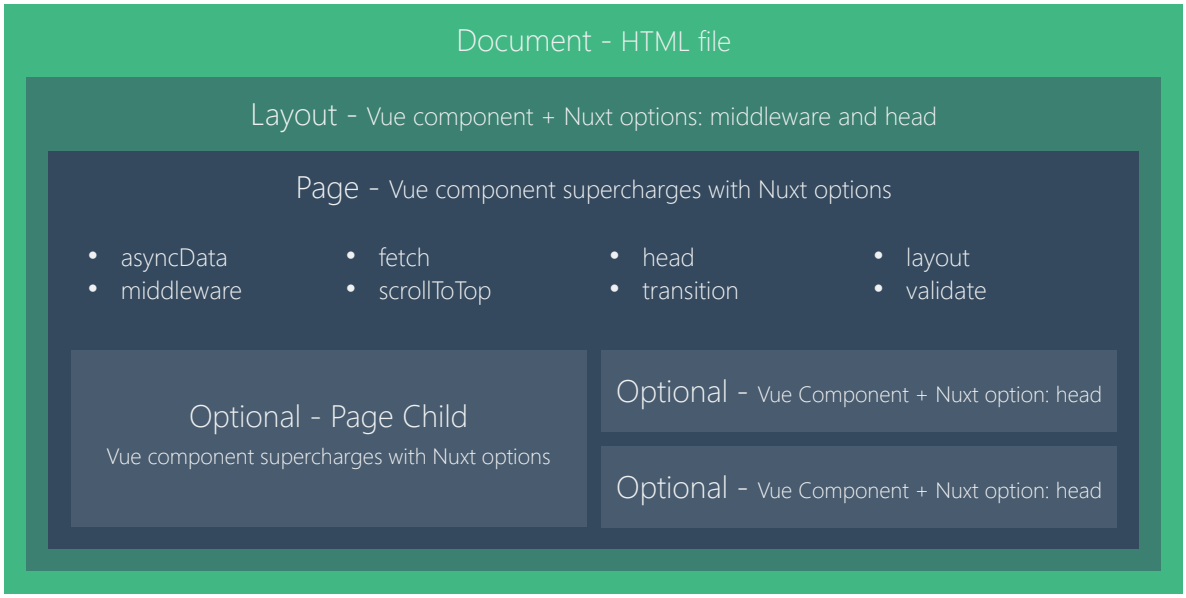
配置路由

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`

> win10有时需管理员权限启动vscode

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>
```

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']
},
```

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

Vue3初体验

知识点

- 调试环境搭建
- 源码结构
- Composition API
- 数据响应式革新
- vue3展望

调试环境搭建

- 迁出Vue3源码：`git clone https://github.com/vuejs/vue-next.git`
- 安装依赖：`yarn`
- 添加SourceMap文件：
  - rollup.config.js

    ```
    // 76行添加如下代码
    output.sourcemap = true
    ```

  - 修改ts配置，tsconfig.json

    ```
    "sourceMap": true
    ```

- 编译：`yarn dev`

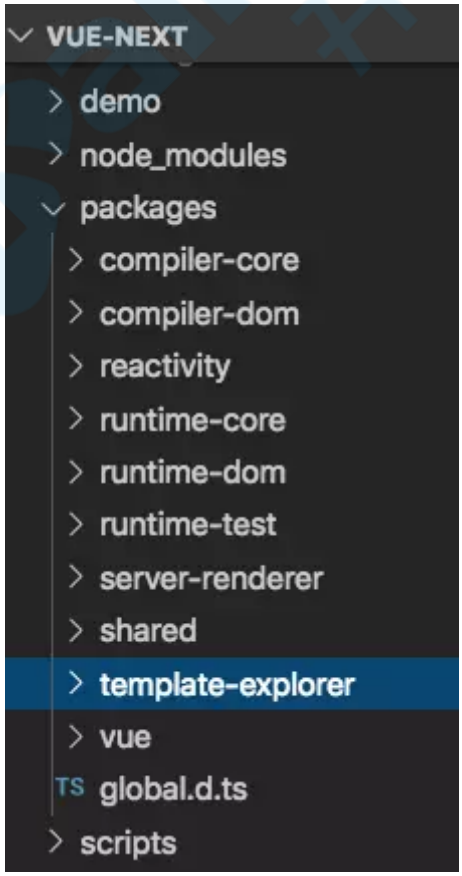
  > 生成结果：packages\vue\dist\vue.global.js

- 测试代码，~/samples/01-hello.html

```html
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>hello vue3</title>
    <script src="../packages/vue/dist/vue.global.js"></script>
</head>

<body>
    <div id='app'></div>
    <script>
        const App = {
            template: `<h1>{{message}}</h1>`,
            data: { message: 'Hello Vue 3!' }
        }
        Vue.createApp().mount(App, '#app')
    </script>
</body>

</html>
```

好了 同学们可以快乐的玩耍了。

源码结构



源码位置是在package文件件内，实际上源码主要分为两部分，编译器和运行时环境。

- 编译器
  - compiler-core 核心编译逻辑
    - 模板解析
    - AST
    - 代码生成
  - compiler-dom 针对浏览器的编译逻辑
    - v-html
    - v-text
    - v-model
- 运行时环境
  - runtime-core 运行时核心
    - inject
    - 生命周期
    - watch
    - directive
    - component
  - runtime-dom 运行时针对浏览器的逻辑
    - class
    - style
  - runtime-test 测试环境仿真，主要为了解决单元测试问题的逻辑 在浏览器外完成测试比较方便
- reactivity 响应式逻辑
- template-explorer 模板解析器 可以这样运行

```
yarn dev template-explorer
```

- vue 代码入口，整合编译器和运行时
- server-renderer 服务器端渲染（TODO）
- share 公用方法

Composition API

Composition API字面意思是组合API，它是为了更方便的实现逻辑的组合而产生的。

- 主要api如下

```
const {
    createApp,
    reactive, // 创建响应式数据对象
    ref, // 将单个值包装为一个响应式对象
    toRefs, // 将响应式数据对象转换为单一响应式对象
    computed, // 创建计算属性
    watch, // 创建watch监听
    // 生命周期钩子
    onMounted,
    onUpdated,
    onUnmounted,
} = Vue
```

- 基本使用

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>Document</title>
    <script src="../packages/vue/dist/vue.global.js"></script>
</head>

<body>
    <div id="app"></div>
    <script>
        const { createApp, reactive } = Vue
        const App = {
            template: `<h1>{{ state.message }}</h1>`,
            // setup函数会在 beforeCreate之后 created之前执行
            setup() {
                // 使用reactive做响应化处理
                const state = reactive({ message: 'Hello Vue 3!' });
                // 返回上下文
                return {state};
            }
        }
        createApp().mount(App, '#app')
    </script>
</body>
</html>
```

> 观察setup参数:
>
> ```
> setup(props,context){
>     console.log('setup....',)
>     console.log('props',props) // 组件参数
>     console.log('context',context) // 上下文对象
> }
> ```

- 事件处理

```
const App = {
    template: `<h1 @click="onClick">{{ state.message }}</h1>`,
    setup() {
        // 声明回调
        function onClick() {
            state.message = 'Vue3初体验! ';
        }
        // 添加到上下文
        return {state, onClick};
    }
}
```

- 生命周期

```
const { onMounted } = Vue

const App = {
    setup() {
       onMounted(() => {
            console.log('App挂载! ');
       })
    }
}
```

- 单值响应化：ref

```
const { ref } = Vue
const App = {
    template: `<h1 @click="onClick">{{ message }}</h1>`,
    setup() {
        // ref返回包装对象
        const message = ref('Hello Vue 3!');
        function onClick() {
            // 包装对象要修改其value
            message.value = 'Vue3初体验! ';
        }
        // 指定包装对象到上下文，模板中可以直接用
        return { message, onClick };
    }
}
```

- toRefs：将reactive创建出的对象展开为基础类型

```
const { toRefs } = Vue
const App = {
    // ...
    setup() {
        // 转换每个属性并展开
        return { ...toRefs(state), onClick };
    }
}
```

- computed：计算属性

```
const { createApp, reactive, ref, toRefs, onMounted, computed } = Vue
const App = {
    template: `
        <h1 @click="onClick">{{ message }}</h1>
        <p>{{msg}}</p>
    `,
    setup() {
        // 使用reactive做响应化处理
        const state = reactive({
            msg: computed(() => 'computed: ' + state.message)
        });
    }
}
```

- watch：创建监控表达式

```
// 参数1是监控表达式创建函数，其他的雷同
watch(() => state.message, (val, oldVal) => {
    console.log(`message变了，新值是：${val}`)
})
```

- 体验逻辑组合

```
const { onMounted, onUnmounted } = Vue

// 鼠标位置侦听
function useMouse() {
    const state = reactive({ x: 0, y: 0 })
    const update = e => {
        state.x = e.pageX
        state.y = e.pageY
    }
    onMounted(() => {
        window.addEventListener('mousemove', update)
    })
    onUnmounted(() => {
        window.removeEventListener('mousemove', update)
    })
    return toRefs(state)
}
// 事件监测
function useTime() {
    const state = reactive({ time: '' })
    onMounted(() => {
        setInterval(() => {
            state.time = new Date()
        }, 1000)
    })
    return toRefs(state)
}
// 逻辑组合
const MyComp = {
    template: `<div>x:{{ x }} y:{{ y }} z:{{ time }} </div>`,
```

```
    setup() {
        // 使用鼠标逻辑
        const { x, y } = useMouse()
        // 使用时间逻辑
        const { time } = useTime()

        return { x, y, time }
    }
}
```

数据响应式革新

- Vue2响应式的一些问题：
  - 当data/computed/props中数据规模庞大，遍历起来会很慢，要监听所有属性的变化，占用内存会很大
  - 无法监听Set/Map的变化；Class类型的数据无法监听；属性新加或删除无法监听；数组元素增加和删除无法监听；对于数组需要额外实现方法拦截，对应的修改语法也有限制。
- Vue3响应式原理：使用ES6的Proxy来解决这些问题。

```
function reactive(data) {
    if (typeof data !== 'object' || data === null) {
        return data
    }
    // Proxy相当于在对象外层加拦截
    // http://es6.ruanyifeng.com/#docs/proxy
    const observed = new Proxy(data, {
        // 获取拦截
        get(target, key, receiver) {
            // Reflect用于执行对象默认操作，Proxy的方法它都有对应方法
            // Reflect更规范、更友好
            // http://es6.ruanyifeng.com/#docs/reflect
            const val = Reflect.get(target, key, receiver);
            // 若val为对象则定义代理
            return typeof val === 'object' ? reactive(val) : val;
        },
        // 新增、更新拦截
        set(target, key, value, receiver) {
            effective()
            return Reflect.set(target, key, value, receiver)
        },
        // 删除属性拦截
        deleteProperty(target, key){
            return Reflect.deleteProperty(target,key)
        }
    })
    return observed
}
```

测试代码

```
const data = {
  foo: 'foo',
```

```
  obj: {a:1},
  arr: [1,2,3]
}
const react = reactive(data)
// get
react.foo
react.obj.a
react.arr[0]

// set
react.foo = 'fooooo'
react.obj.a = 10
react.arr[0] = 100

// add
react.bar = 'bar'
react.obj.b = 10
react.arr.push(4)
react.arr[4] = '5'

// delete
delete react.bar
delete react.obj.b
react.arr.splice(4, 1)
delete react.arr[3]
```

vue3展望

- vue3会兼容之前写法，仅少量Portal、Suspense之类的组件需要看看，composition可选
- 正式版发布还有一段时间，相关工具、生态、库都跟不上，vue3也许明年都不会有大需求
- vue3重要特性如响应式会给大数据量应用带来福音，建议跟上；time-slicing对于用户交互敏感的应用是重要特性，比如证券类应用；既然性能号称很好，移动端应用也更加适合使用了。
- 虽然不是正式版，但仍可提前学习vue3设计和实现思想

学习资源分享

开课吧大圣vue3剖析

开课吧老夏vue3尝鲜

开课吧老杨的直播间