Umi 中如何根据服务端响应数据动态更新路由

路由系统是 Umi 很重要的一个部分，常规用法中我们最常使用或者听到的就是所谓的“约定式”和“配置式”， 但其实这两者只是通过显式或者隐式的方式获取到“路由配置”，将他传给 `react-router-dom` 渲染。

比如，有如下文件目录结构：

```
+ pages/
  + users/
    - index.ts
  - index.ts
```

Umi 中的路由渲染流程

约定式

将会通过约定式规则生成如下路由配置:

```json
[
    { path: '/', component: 'index' },
    { path: '/users', component: 'users/index' },
],
```

配置式

如果你将上述的“路由配置”写到 umi 的配置文件中，比如：

```ts
export default {
  routes: [
    { path: '/', component: 'index' },
    { path: '/users', component: 'users/index' },
  ],
}
```

即表示，你是使用配置式路由，而约定式的规则将会被停用，也就是后续你新增页面文件之后，你也需要手动来增加修改 `routes` 配置。

react-router 渲染

然后 Umi 会使用这一份路由配置，生成如下的组件格式：

```tsx 复制代码
// 演示作用的伪代码
import React from 'react';
import { Route, Router, Routes, useRoutes } from 'react-router-dom';

const Home = React.lazy(() => import('src/pages/index.tsx'));
const Users = React.lazy(() => import('src/pages/users/index.tsx'));

function Page(props: any) {
  return (
    <Router>
      <Routes>
          <Route path="/" element={<Home />} />
          <Route path="/users" element={<Users />} />
      </Routes>
    </Router>
  );
}
```

react-router 渲染中的路由更新

简单理解了上面的流程和代码之后，在回过头来看我们的问题 - “根据服务端响应数据动态更新路由”。 如果你把它当作一个普通组件来处理，简单的修改上面的代码：

```tsx 复制代码
// 演示作用的伪代码
import React from 'react';
import { Route, Router, Routes, useRoutes } from 'react-router-dom';

const Home = React.lazy(() => import('src/pages/index.tsx'));
const Users = React.lazy(() => import('src/pages/users/index.tsx'));

function Page(props: any) {
  const data = fetch('http://example.com/api/routes.json');

  return (
    <Router>
      <Routes>
```

```
          <Route path="/" element={<Home />} />
          <Route path="/users" element={<Users />} />
          {data.map(i=>{
            return (<Route path={i.path} element={i.type==='users'?<Users />:<Home />} />)
          })}
        </Routes>
      </Router>
    );
}
```

细心的你，应该能从上面的代码中，看到两个关键点：

1、请求需要在 render 之前发起

2、组件需要本地编译，服务端只能指定页面的类型

第一点比较好理解，我们先来说说第二点最常遇到的问题：使用配置式路由，动态添加路由数据，开发的时候是好的，到生产会找不到页面的文件。 这是因为 umi dev 的时候，本地的 pages 文件都会被动态构建，在开发的时候，你就能够找到文件并匹配上，而到了生产 umi build 的时候，则会因为 treeshaking 的作用，静态分析的时候，没使用到的 pages 文件，将会从构建产物总被剔除。

现在我们先回归一下 “Umi 中如何根据服务端响应数据动态更新路由”?

Umi 中如何根据服务端响应数据动态更新路由

静态添加路由

相信只要你查找过这个问题相关的信息，你很容易就能找到官网上的用例：

scss 复制代码

```
patchClientRoutes({ routes })

修改被 react-router 渲染前的树状路由表，接收内容同 useRoutes。

比如在最前面添加一个 /foo 路由，
```

tsx 复制代码

```
import Page from '@/extraRoutes/foo';

export function patchClientRoutes({ routes }) {
  routes.unshift({
```

```
    path: '/foo',
    element: <Page />,
  });
}
```

还是上诉提到的路由文件结构，但是我们在运行时配置中增加 patchClientRoutes:

tsx 复制代码

```tsx
import Page from "@/pages/users";

export function patchClientRoutes({ routes }) {
  routes.unshift({
    path: "/foo",
    element: <Page />,
  });
}
```

`umi dev` 之后，访问 `http://localhost:8001/foo` 确实能够访问到 `Users` 页面。

看起来这似乎是一个非常简单的问题啊，那我们接着往下看文档

动态添加路由

复制代码

```
比如和 render 配置配合使用，请求服务端根据响应动态更新路由，
```

tsx 复制代码

```tsx
let extraRoutes;

export function patchClientRoutes({ routes }) {
  // 根据 extraRoutes 对 routes 做一些修改
  patch(routes, extraRoutes);
}

export function render(oldRender) {
  fetch('/api/routes')
    .then((res) => res.json())
    .then((res) => {
      extraRoutes = res.routes;
      oldRender();
    });
}
```

到这里很多朋友就很懵逼了，fetch 返回值类型是啥？`patch` 方法写了啥？`extraRoutes` 干啥用的，为什么要多引入一个对象？

为什么作为 Umi 的官方文档，要写成这样？谁实话这里的问的确实可以补充的更详细一点，比如把这个用例再完善一下，但其实这么写的用意是，这里可以支持任意的服务端返回内容，只要你根据服务端返回数据，修改对应的处理过程即可。

`extraRoutes` 的作用，其实就是上面我们提到的 `请求需要在 render 之前发起`。

下面我列一个最简单的用例：

首先增加一个 mock 数据：

ts 复制代码

```ts
// 新建 mock/app.ts
export default {
  "/api/routes": {
    routes: [
      { path: "/foo", type: "home" },
      { path: "/users", type: "users" },
    ],
  },
};
```

然后修改运行时配置 app.ts：

tsx 复制代码

```tsx
import React from "react";

const Home = React.lazy(() => import("@/pages/index"));
const Users = React.lazy(() => import("@/pages/users"));

let extraRoutes: any[];

export function patchClientRoutes({ routes }) {
  extraRoutes.forEach((route) => {
    routes.unshift({
      path: route.path,
      element: route.type === "users" ? <Users /> : <Home />,
    });
  });
}

export function render(oldRender) {
  fetch("/api/routes")
    .then((res) => res.json())
    .then((res) => {
      extraRoutes = res.routes;
      oldRender();
```

```
    });
}
```

auth

当然大多时候，我们做动态添加路由的目的，就是不同的用户可以访问不同的页面。在这里我们加入模拟的授权检测机制

修改接口数据

tsx 复制代码

```tsx
// 用来记录用户是否登录的全局对象。一般是记录在 cookie 里面。
if (!global.auth) {
  global.auth = false;
}

const getAuth = () => {
  return global.auth;
};

export default {
  "/api/routes": {
    routes: [
      { path: "/foo", type: "home" },
      { path: "/users", type: "users" },
    ],
  },
  "/api/checkLogin": async (req, res) => {
    res.send({
      isAuth: getAuth(),
    });
  },
  "/api/login": async (req, res) => {
    global.auth = true;
    res.send({
      success: true,
    });
  },
};
```

增加一个登录页面 `src/pages/login/index.tsx`

tsx 复制代码

```tsx
import React from "react";

const Page = () => (
```

```tsx
  <div>
    <button
      onClick={() => {
        fetch("/api/login")
          .then((res) => res.json())
          .then((res) => {
            if (res.success) {
              window.location.reload();
            }
          });
      }}
    >
      Click Me For Mock Login
    </button>
  </div>
);

export default Page;
```

将登录页面写到路由配置中

tsx 复制代码

```tsx
// .umirc.ts
export default {
    routes: [
      { path: '/', component: '@/pages/index' },
      { path: '/login', component: '@/pages/login' },
    ],
  }
```

最终修改运行时配置

tsx 复制代码

```tsx
import React from "react";
import { history } from "umi";
const Home = React.lazy(() => import("@/pages/index"));
const Users = React.lazy(() => import("@/pages/users"));

let extraRoutes: any[] = [];

export function patchClientRoutes({ routes }) {
  extraRoutes.forEach((route) => {
    routes.unshift({
      path: route.path,
      element: route.type === "users" ? <Users /> : <Home />,
    });
  });
}
```

```
export function render(oldRender) {
  fetch("/api/checkLogin")
    .then((res) => res.json())
    .then((res) => {
      if (res.isAuth) {
        fetch("/api/routes")
          .then((res) => res.json())
          .then((res) => {
            extraRoutes = res.routes;
            const urlParams = new URL(window.location.href).searchParams;
            const redirect = urlParams.get("redirect");
            if (redirect) {
              history.push(redirect);
            }
            oldRender();
          });
      } else {
        history.push(
          "/login?redirect=" + window.location.pathname + window.location.search
        );
        oldRender();
      }
    });
}
```

umi3 兼容

如果你在 Umi@3 的时候，就已经用过 patchRoutes 来动态添加路由，那你需要做以下修改

1、将 patchRoutes 修改为 patchClientRoutes 这两者只是改名了。 Umi@4的 patchRoutes 有其他的用处

2、将之前 routes 里面的 `item.component` 取出来，放到 element 中，将之前 route 配置中的其他数据，取出来手动传到 element 中，如：

tsx 复制代码

```
routes.map(item => {
      const Component = item.component;
      const { component, ...restItem } = item;
      return ({
        ...item,
        element: <Component defaultProps={restItem} {...item} />,
        defaultProps: restItem,
```

```
    });
  })
```

3、如果你之前页面组件是封装到子包中的，现在要同时用于 Umi3 和 Umi4 ，并且你用到 如 `props.xxx` 的一些 Umi@3 内置传递给组件的参数，那你可以在 Umi@4 项目中使用 `withRouter` 包裹你的组件，如：

tsx 复制代码

```tsx
import React from "react";
import { withRouter } from "umi";
const Users = React.lazy(() => import("@/pages/users"));

export function patchClientRoutes({ routes }) {
  const Foo = withRouter(Users);
  routes.unshift({
    path: "/foo",
    element: <Foo />,
  });
}
```

> 如果你希望在 umi@4 的页面中依旧能用到 props.location 之类的参数，你可以开启兼容配置,如：
>
> ts 复制代码
>
> ```ts
> export default {
>   reactRouter5Compat: {},
>   historyWithQuery: {},
> };
> ```