Next.js 全解（下）

Node.js Web 系列课程

版权声明

联系方式

如果你想要购买本课程
请微信联系 xiedaimala02 或 xiedaimala03

如果你发现有人盗用本课程
请微信联系 xiedaimala02 或 xiedaimala03

回顾一下

- 创建项目
  - ✓ npm init next-app 项目名
- 快速导航
  - ✓ <Link href=xxx> 包住 <a>
- 同构代码
  - ✓ 一份代码，两端运行
- 全局组件
  - ✓ pages/_app.js
- 自定义 head
  - ✓ 使用 <Head> 组件
- 全局 CSS
  - ✓ 在 _app.js 里 import
- 局部 CSS
  - ✓ <style jsx>
  - ✓ <style jsx global> 不推荐
  - ✓ xxx.module.css
- SCSS
  - ✓ 安装 sass 依赖即可

启用 TypeScript

- 创建 tsconfig.json
  - ✓ tsc --init 运行后得到 tsconfig.json
  - ✓ 将 jsconfig.json 里面的配置合并到 tsconfig.json
  - ✓ 删除 jsconfig.json
- 重启 yarn dev
  - ✓ yarn add --dev typescript @types/react @types/node
  - ✓ yarn dev
- 改后缀
  - ✓ 将文件名由 .js 改为 .tsx
  - ✓ 不需要一次将所有文件全部改完

tsconfig 加强

- 在 tsconfig.json 里添加
  - ✓ "noImplicitAny": true
  - ✓ 禁用隐式的 any

Next.js API

- 目前的页面
  - ✓ index 和 posts/first-post 都是 HTML
  - ✓ 但实际开发中我们需要请求 /user /shops 等 API
  - ✓ 返回的内容是 JSON 格式的字符串
- 使用 Next.js API
  - ✓ 路径为 /api/v1/posts 以便与 /posts 区分开来
  - ✓ 默认导出的函数的类型为 NextApiHandler
  - ✓ 该代码只运行在 Node.js 里，不运行在浏览器中

/pages/api/v1/posts.tsx

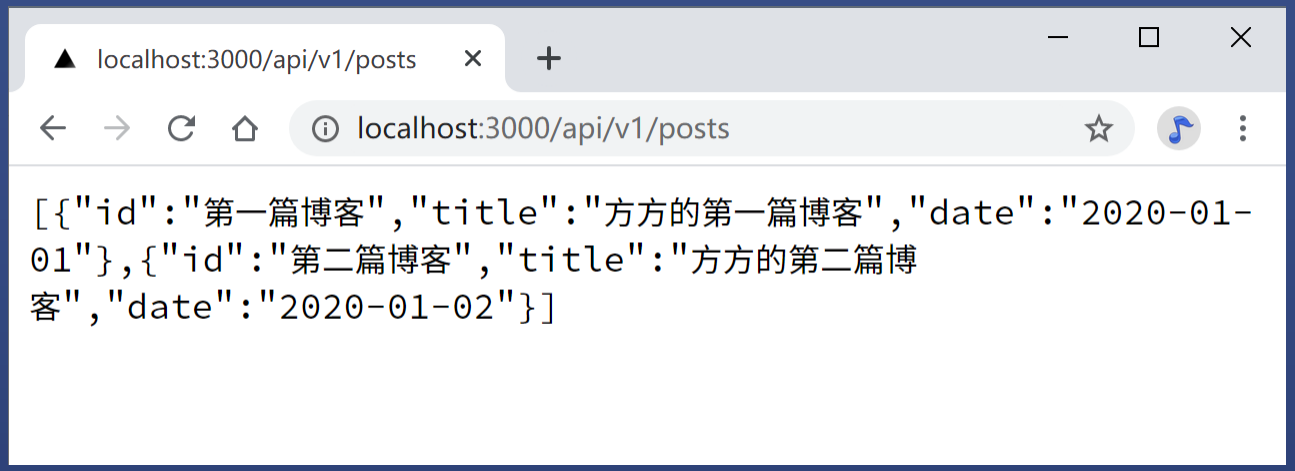
```
const posts: NextApiHandler = async (req, res)
=> {
  res.statusCode = 200;
  res.setHeader('Content-Type',
'application/json');
  const posts = await getPosts();
  res.end(JSON.stringify(posts));
};

export default posts;
```

lib/posts.tsx

```
export const getPosts = async () => {
  const fileNames = await
fsPromise.readdir(find('markdown'))
  return fileNames.map(fileName => {
    const id = fileName.replace(/\.md$/, '')
    const fullPath = find('markdown', fileName)
    const content = fs.readFileSync(fullPath, 'utf-8')
    const {title, date} = matter(content).data
    return {
      id, title, date
    }
  })
}
```

Next.js API


localhost:3000/api/v1/posts
localhost:3000/api/v1/posts
[{"id":"第一篇博客","title":"方方的第一篇博客","date":"2020-01-01"},{"id":"第二篇博客","title":"方方的第二篇博客","date":"2020-01-02"}]

API 小结

- /api/ 里的文件是 API
  - 一般返回 JSON 格式的字符串
  - 但也不是不能返回 HTML，比如 res.end('<h1>...
- API 文件默认导出 NextApiHandler
  - 这是一个函数类型
  - 第一个参数是请求
  - 第二个参数是对象
  - 咦？怎么没有 next()，怎么做中间件呢？
  - Next.js 基于 Express，所以支持 Express 的中间件
  - 这个以后再教，你现在可以先看文档。

Next.js 三种渲染

- 客户端渲染
  - ✓ 只在浏览器上执行的渲染
- 静态页面生成（SSG）
  - ✓ Static Site Generation，解决白屏问题、SEO问题
  - ✓ 无法生成用户相关内容（所有用户请求的结果都一样）
- 服务端渲染（SSR）
  - ✓ 解决白屏问题、SEO 问题
  - ✓ 可以生成用户相关内容（不同用户结果不同）

- ✓ 注意：SSR 和 SSG 都属于预渲染 Pre-rendering

旧瓶装新酒

- 三种渲染方式分别对应
  - ✓ 客户端渲染——用JS、Vue、React 创建HTML
  - ✓ SSG——页面静态化，把 PHP 提前渲染成 HTML
  - ✓ SSR——PHP、Python、Ruby、Java 后台的基本功能
- 不同点
  - ✓ Next.js 的预渲染可以与前端 React 无缝对接，下文有讲

客户端渲染

- 用浏览器 JS 创建 HTML
  - ✓ 给大家表演一个加载 posts
- 要点
  - ✓ 如何封装 usePosts，注意过程
- 文件列表
  - ✓ pages/posts/index.tsx
  - ✓ lib/hooks/usePosts.tsx
- 总结
  - ✓ 文章列表是完全由前端渲染的，我们称之为客户端渲染

客户端渲染的缺点

- 白屏
  - ✓ 在 AJAX 得到响应之前，页面中之后 Loading
- SEO 不友好
  - ✓ 搜索引擎访问页面，看不到 posts 数据
  - ✓ 因为搜索引擎默认不会执行 JS，只能看到 HTML

静态内容 v.s. 动态内容

```
const PostsIndex: NextPage = (data) => {
  const {isLoading, posts, isEmpty} = usePosts();
  return (
    <div>
      <h1>文章列表</h1>
      {
        isLoading ? <div>加载中...</div> :
          isEmpty ? <div>目前没有文章</div> :
            <>
              {posts?.map(p =>
                <div key={p.id}>{p.id}</div>
              )}
            </>
      }
    </div>
  );
};
```

一般来说，静态内容是写在代码里的，动态内容是来自数据库的

这节课我们用文件系统代替数据库

思考题

- 上图中的静态内容
  - 是服务端渲染的，还是客户端渲染的?
  - 渲染了几次？一次还是两次?
- 参考 React SSR 的官方文档
  - 推荐在后端 renderToString() 在前端 hydrate()
  - hydrate() 混合，会保留 HTML 并附上事件监听
  - 也就是说后端渲染 HTML，前端添加监听
  - 前端也会渲染一次，用以确保前后端渲染结果一致
  - 看视频中的分析
- 推论
  - 所有页面至少有一个标签是静态内容，由服务器渲染

静态页面生成(SSG)

- 背景
  - 你有没有想过，其实每个人看到的文章列表都是一样的
  - 那么为什么还需要在每个人的浏览器上渲染一次
  - 为什么不在后端渲染好，然后发给每个人
  - N 次渲染变成了 1 次渲染
  - N 次客户端渲染变成了 1 次静态页面生成
  - 这个过程叫做动态内容静态化
- 思考
  - 显然，后端最好不要通过 AJAX 来获取 posts（为什么）
  - 那么，应该如何获取 posts 呢?
  - 补充：如果后端想通过 AJAX 获取 posts 也不是不可以

getStaticProps 获取 posts

- 声明位置
  - ✓ 每个 page 不是默认导出一个函数么？
  - ✓ 把 getStaticProps 声明在这个函数旁边即可
  - ✓ 别忘了加 export
- 写法

```
export const getStaticProps = async () => {
  const posts = await getPosts();
  return {props: {posts: posts}};
};
```

高亮部分是固定的，不能变的。

getStaticProps

- 如何使用 props

```
export default function PostsIndex = (props)
=> { ... }
```

默认导出的函数的第一个参数就是 props

- 如何给 props 添加类型

```
const PostsIndex: NextPage<{ posts: Post[] }>
= (props) => { ... }
```

- ✓ 把 function 改成 const + 箭头函数
- ✓ 类型声明为 NextPage
- ✓ 用泛型给 NextPage 传个参数 <Props>
- ✓ Props 就是 props 的类型

同构！

- 有没有发现
  - ✓ 现在前端不用 AJAX 也能拿到 posts 了！
  - ✓ 这就是同构 SSR 的好处：后端数据可以直接传给前端
  - ✓ 前端 JSON.parse 一下就能得到了 posts（帮你做了）
- 难道 PHP/Java/Pyton 就做不到么
  - ✓ 其实也可以做到，思路一样
  - ✓ 但是它们不支持 JSX，很难与 React 无缝对接
  - ✓ 而且它们的对象不能直接提供给 JS 用，需要类型转换

静态化的时机

- 环境
  - ✓ 在开发环境，每次请求都会运行一次 getStaticProps
  - ✓ 这是为了方便你修改代码重新运行
  - ✓ 在生产环境， getStaticProps 只在 build 时运行一次
  - ✓ 这样可以提供一份 HTML 给所有用户下载
- 如何体验生产环境
  - ✓ 关掉 yarn dev
  - ✓ yarn build
  - ✓ yarn start

生产环境

- 解读
  - λ (Server) SSR 不能自动创建 HTML (等会再说)
  - ◯ (Static) 自动创建 HTML (发现你没用到 props)
  - ● (SSG) 自动创建 HTML JS JSON (发现你用到了 props)
- 三种文件类型

posts.html 含有静态内容，用于用户直接访问
posts.js 也含有静态内容，用于快速导航（与 HTML 对应）
posts.json 含有数据，跟 posts.js 结合得到界面

- 为什么不直接把数据放入 posts.js 呢?
- 显然，是为了让 posts.js 接受不同的数据（下文解释）
- 当然，目前只能接受一个数据（来自 getStaticProps）

小结

- 动态内容静态化
  - ✓ 如果动态内容与用户无关，那么可以提前静态化
  - ✓ 通过 getStaticProps 可以获取数据
  - ✓ 静态内容 + 数据（本地获取）就得到了完整页面
  - ✓ 代替了之前的静态内容 + 动态内容（AJAX 获取）
- 时机
  - ✓ 静态化是在 yarn build 的时候实现的
- 优点
  - ✓ 生产环境中直接给出完整页面
  - ✓ 首屏不会白屏
  - ✓ 搜索引擎能看到页面内容（方便 SEO）

如果页面跟用户相关呢?

比如根据用户的 user id 显示不同的信息流

用户相关动态内容

- 较难提前静态化
  - 需要在用户请求时，获取用户信息，然后通过用户信息去数据库拿数据
  - 如果硬要做，就要给每个用户创建一个页面
  - 有时候这些数据更新极快，无法提前静态化
  - 比如微博首页的信息流
- 那怎么办？
  - 要么客户端渲染，下拉更新(1)
  - 要么服务端渲染，下拉更新(2)
  - 但这次的服务端渲染不能用 getStaticProps
  - 因为 getStaticProps 是在 build 时执行的
  - 可用 getServerSideProps(context: NextPageContext)

getServerSideProps

- 运行时机
  - ✓ 无论是开发环境还是生产环境
  - ✓ 都是在请求到来之后运行 getServerSideProps
- 回顾一下 getStaticProps
  - ✓ 开发环境，每次请求到来后运行，方便开发
  - ✓ 生产环境，build 时运行一次
- 参数
  - ✓ context，类型为 NextPageContext
  - ✓ context.req / context.res 可以获取请求和响应
  - ✓ 一般只需要用到 context.req

示例

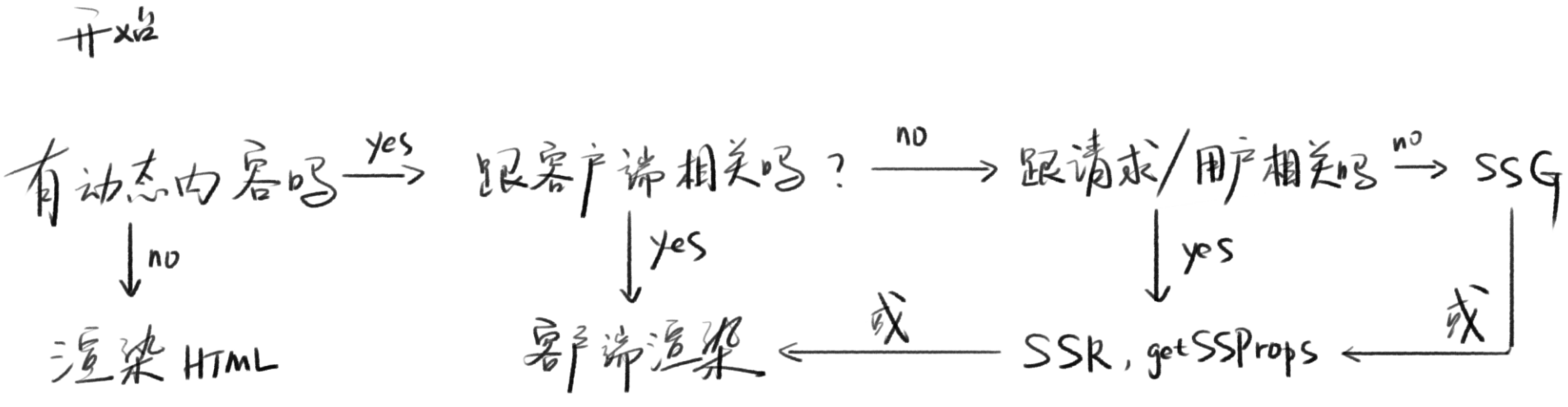
```tsx
const index: NextPage<Props> = (props) => {
  const {browser} = props;
  return (
    <div>
      <h1>你的浏览器是 {browser.name}</h1>
    </div>
  );
};
export default index;
```

- ✓ 展示了当前用户的浏览器
- ✓ 这些信息不可能在请求之前知道
- ✓ 思考：如果我要在页面上展示当前窗口大小，可以吗
- ✓ 答案：只能用客户端渲染做到

总结

- 静态内容
  - ✓ 直接输出 HTML，没有术语
- 动态内容
  - ✓ 术语：客户端渲染，通过 AJAX 请求，渲染成 HTML
- 动态内容静态化
  - ✓ 术语：SSG，通过 getStaticProps 获取用户无关内容
- 用户相关动态内容静态化
  - ✓ 术语：SSR，通过 getServerSideProps 获取请求
  - ✓ 缺点：无法获取客户端信息，如浏览器窗口大小

流程图



- 有动态内容吗？没有什么都不用做，自动渲染为 HTML
- 动态内容跟客户端相关吗？相关就只能用客户端渲染（BSR）
- 动态内容跟请求/用户相关吗？相关就只能用服务端渲染（SSR）或 BSR
- 其他情况可以用 SSG 或 SSR 或 BSR

还差一个功能

- 点击 posts 列表查看文章
  - 简单，不就是加个 Link>a 标签吗
  - href=`/posts/${id}`
- 新建的文件名应该叫做什么
  - pages/posts/[id].tsx
  - 你没有看错，文件名就是 [id].tsx
- /pages/posts/[id].tsx 的作用
  - 既声明了路由 /posts/:id
  - 又是 /posts/:id 的页面实现程序
  - 这么做，真是妙啊

[id].tsx

- 步骤
  - ✓ 实现 PostsShow，从 props 接收 post 数据
  - ✓ 实现 getStaticProps，从第一个参数接受 params.id
  - ✓ 实现 getStaticPaths，返回 id 列表
- 优化
  - ✓ 使用 marked 得到 markdown 的 HTML 内容
- build
  - ✓ 中断 yarn dev
  - ✓ yarn build 然后看一下 .next/server 目录
  - ✓ yarn start

填坑

- 接受不同的 json 数据
  - 回到 PPT
- fallback: false 的作用
  - 是否自动兜底
  - false 表示如果请求的 id 不在 getStaticPaths 的结果里，则直接返回 404 页面
  - true 表示自动兜底，id 找不到依然渲染页面
  - 注意 id 不在结果里不代表 id 不存在，比如大型项目无法讲所有产品页面都静态化，只静态化部分 id 对应的页面

总结

- 制作 API
  - ✓ 放在 /pages/api/v1 目录里
- 三种渲染方式
  - ✓ BSR / SSG / SSR
  - ✓ 记住我画的流程图
- 三个 API
  - ✓ getStaticProps (SSG)
  - ✓ getStaticPaths (SSG)
  - ✓ getServerSideProps (SSR)
- 概念
  - ✓ 白屏是为什么
  - ✓ BSR 为什么不适合 SEO
  - ✓ 静态化是什么
  - ✓ 同构是什么