ChatGPT

Next.js 15 & React 19 全栈项目中的 TypeScript 类型最佳实践

1. `type` 与 `interface` 的区别及适用场景

- **核心区别**：`interface` 主要用于描述对象的形状和契约，支持继承（`extends`）和声明合并；`type` 则能定义任意类型（原始类型、联合类型、交叉类型、映射类型等）[1][2]。二者编译后都被移除，不影响运行时性能[3]，选择时以可读性和扩展性为准。
- **组件 Props**：推荐使用 `interface` 定义组件 Props，因为它直观地描述了对象结构，并且方便扩展。例如：

```
interface ButtonProps {
  label: string
  onClick: () => void
}
function Button({ label, onClick }: ButtonProps) { ··· }
```

  如 Daniel Leitch 建议，在 React 中「从 `interface` 开始，考虑到其可扩展性和对象形状的清晰表述」[2]。当然简单场景下用 `type` 定义 Props 也可以，但如果需要后续扩展，`interface` 更加灵活（可继承、可声明合并）。
- **API 请求/响应模型**：如果返回的数据结构较固定，`interface` 可清晰表达；但很多 API 模型需要联合类型、映射或条件类型时，`type` 更方便。例如使用 Zod 定义请求参数时，经常用 `z.infer` 生成 `type`：

```
const CreatePostSchema = z.object({ title: z.string(), content: z.string() })
export type CreatePostInput = z.infer<typeof CreatePostSchema>
```

  在定义多种可能的响应或联合类型时，`type` 也更加灵活。一般建议对数据契约使用 `interface`，对复杂类型（联合、交叉、元组等）使用 `type`[4][5]。
- **数据库模型（Prisma）**：Prisma 会根据 schema 自动生成 TS 类型（通常为类型别名）。我们通常不直接修改生成类型，而是基于它们通过工具类型定义 API 的 DTO。例如，为 `User` 模型创建“新建用户输入”可以这样写：

```
import type { User } from '@prisma/client'
// 排除主键、时间戳等字段，定义创建用户的输入类型
type CreateUserInput = Omit<User, 'id' | 'createdAt' | 'updatedAt'>
```

  如 Prisma 团队建议，业务层的请求/响应类型最好与数据库模型分离、各自维护[6]。可以用 TS 自带的 `Omit`、`Pick` 等工具类型从 Prisma 生成的类型派生出适合前后端交互的类型[7]。例如上面例子引用了 Prisma 生成的 `User` 类型，再用 `Omit` 去掉不需要的字段，得到干净的输入类型[7]。

2. 全栈项目中的类型共享

在 Next.js 全栈项目中，前后端代码通常在同一个代码库内，TypeScript 类型可以在 Server Components、Server Actions、API 路由和客户端共享，避免重复定义。

- **共用类型模块**：可以把公共类型定义放在单独的模块（如 `lib/types.ts`、`types/` 文件夹）中，Server 和 Client 都可导入使用。由于 TS 类型在编译后会被删除，Next.js 会剔除未被引用的服务端代码 [8]，因此在客户端引用类型定义（甚至导入带有服务端标识的文件）不会污染最终打包包大小。正如讨论中提到的："Next.js（或 Webpack）只会打包被实际引用的代码。即使从服务端文件导入，只要不在客户端真正使用，它也不会被包含" [8]。因此，我们可以安心地在客户端导入纯类型或仅含类型导出的模块。
- **Server Components 与 Client Components**：React Server Component 和 Client Component 可以引入同一个 TS 类型。只能在服务端使用的代码（如使用 `process.env` 或 Node 专用 API）需要加 `'use server'` 或单独文件标记，但类型定义本身无需关注这些限制即可共享 [8]。例如在服务器组件中定义了返回数据的类型 `type Data = { ... }`，客户端组件只要通过导入相同类型就能保持类型一致，无需在客户端再重复定义。
- **Server Actions**：Next.js 15 引入的 Server Actions（服务器函数）可以在服务器文件中定义并导出，客户端通过 `import` 使用。例如：

```
// app/actions.ts （服务器文件）
'use server'
export async function submitForm(formData: FormData) { ··· }

// app/ui/MyForm.tsx （客户端组件）
'use client'
import { submitForm } from '@/app/actions'
<form action={submitForm}>···</form>
```

因为 `submitForm` 是在服务器标记的文件中定义的，其类型签名 `(formData: FormData) => Promise<void>` 也可以被客户端组件的 Props 类型复用。比如在客户端组件中将这个 action 当作 prop 传递时，可以这样写 Props 类型：

```
interface ClientProps {
  submitAction: (formData: FormData) => void
}
```

此时 `submitAction` 的类型与服务器函数一致，确保端到端类型安全 [9]。

- **tRPC 客户端与服务端**：使用 tRPC 时，服务端定义的路由已经隐式规定了输入输出类型，客户端通过钩子调用时会自动获得这些类型。tRPC 也提供了 `inferRouterInputs` 和 `inferRouterOutputs` 等工具，可以显式提取某个路由的输入输出类型。在应用中，可以这样使用：

```
import type { inferRouterInputs, inferRouterOutputs } from '@trpc/server'
import type { AppRouter } from './server'  // 服务端导出的路由类型

type RouterInput = inferRouterInputs<AppRouter>
type RouterOutput = inferRouterOutputs<AppRouter>

// 提取 post.create 路由的输入和输出类型
```

```
type PostCreateInput = RouterInput['post']['create']
type PostCreateOutput = RouterOutput['post']['create']
```

如 tRPC 文档所示，这样可以在客户端直接推断服务端路由的输入输出类型，保持一致性 [10]。例如在开发表单时，将 `PostCreateInput` 用作表单数据类型，可确保前端传给服务端的参数类型匹配，避免手动重复定义。 [10]

3. 常用 TS 工具类型的应用实例

TypeScript 提供了多种通用工具类型来处理常见场景。以下是一些典型用法和示例代码：

- **`Partial<T>`**：将类型 `T` 的所有属性变为可选（在实际使用时有时仍需部分必填）。常用于表单更新或部分更新请求。

```
interface User {
  id: string
  name: string
  email: string
  password: string
}
// 用户资料更新请求，只需允许修改部分字段
type UpdateUserInput = Partial<Pick<User, 'name' | 'email' | 'password'>>
// 举例：用户可以只更新邮箱
const updateData: UpdateUserInput = { email: 'new@example.com' }
```

在上例中，`UpdateUserInput` 将 `name`、`email`、`password` 全部设为可选，使得前端表单只提交改变的字段成为可能 [11]。`Partial` 还有助于定义默认配置等场景中，允许只写部分属性 [12]。

- **`Pick<T, K>`**：从类型 `T` 中挑选若干属性形成新类型。常用于构造只包含部分字段的 DTO 或视图模型。例如，从 `User` 中提取对外公开的字段：

```
interface User {
  id: string
  name: string
  email: string
  password: string
  isAdmin: boolean
}
// 只返回用户的基本公开信息
type UserPublicInfo = Pick<User, 'id' | 'name' | 'email'>
const publicUser: UserPublicInfo = {
  id: '123', name: 'Alice', email: 'alice@example.com'
}
```

这样可以确保接口只返回所需字段（例如不返回 `password`、`isAdmin` 等敏感或内部字段），符合接口契约 [13]。

- **`Omit<T, K>`**：与 `Pick` 相反，从类型 `T` 中移除若干属性。常用于排除敏感字段或构造“创建时输入”类型。比如不允许客户端传入 `id` 或自动生成的字段：

```
interface Article {
  id: string
  title: string
  content: string
  authorId: string
  createdAt: Date
  updatedAt: Date
}
// 新建文章时不需要客户端提供 id/时间字段
type NewArticleInput = Omit<Article, 'id' | 'createdAt' | 'updatedAt'>
const newPost: NewArticleInput = {
  title: 'New Post', content: 'Hello', authorId: 'user-1'
}
```

  如 Prisma 团队示例，基于 Prisma 的 `User` 模型也可用 `Omit` 构造新用户的输入类型 [7]。在 API 响应中，也可 `Omit<User, 'password'>` 去掉密码字段后再返回给客户端 [14] [7]。

- **`ReturnType<F>`**：提取函数 `F` 的返回类型，对于异步函数通常配合 `Awaited<>` 使用。适用于根据已有函数快速推断类型，避免重复定义。比如：

```
// 假设有一个服务端函数
async function getUser(id: string): Promise<User> {···}
// 从函数类型推断返回类型
type GetUserResult = Awaited<ReturnType<typeof getUser>>  // 即 User 类型
```

  更常见的场景是在 Next.js API 路由或 `fetch` 调用中推断返回的 JSON 结构，以保持前后端一致。例如：

```
// 服务端 API 路由
export async function GET(request: Request) {
  const user: User = ···
  return NextResponse.json(user)
}
// 客户端使用 Fetch，推断响应类型
type UserResponse = Awaited<ReturnType<typeof GET>>  // 推断为 NextResponse<User>
```

  利用 `ReturnType` 可以让前端直接拿到后端函数的返回类型，杜绝复制粘贴错误。

- **其他工具类型**：如 `Required<T>`（将可选属性变必选）、`Record<K,T>`（从键的集合 `K` 构造对象类型）等也常见。举例而言，`Record<string, number>` 表示键为字符串、值为数字的对象，常用于构造配置表或映射表。综合运用这些工具类型，可以大幅减少重复代码并提高类型安全 [15] [14]。

通过合理选用 `type` / `interface`，并充分利用 TypeScript 的工具类型，可以让 Next.js 全栈项目在组件 Props、API 接口、数据库模型之间保持一致且简洁的类型定义，提升开发效率和健壮性 [2] [7]。

参考文献： 上述实践和示例参考了 TypeScript 和 Next.js 官方文档及社区经验 [2] [7] [10] [8] [16]。这些资料介绍了 `type` / `interface` 的特性、tRPC 类型推导方法以及通用工具类型的应用。

---

[1] [5] TypeScript 中，type 和 interface 定义自定义类型的区别 - 炽橙子 - 博客园
https://www.cnblogs.com/ygyy/p/18191941

[2] [3] [4] Choosing Between “type” and “interface” in React | by Daniel Leitch | Nerd For Tech | Medium
https://medium.com/nerd-for-tech/choosing-between-type-and-interface-in-react-da1deae677c9

[6] [7] Using Typescript interfaces with Prisma · prisma prisma · Discussion #14112 · GitHub
https://github.com/prisma/prisma/discussions/14112

[8] Sharing types between server and client · vercel next.js · Discussion #51708 · GitHub
https://github.com/vercel/next.js/discussions/51708

[9] Getting Started: Updating data | Next.js
https://nextjs.org/docs/app/getting-started/updating-data

[10] Inferring Types | tRPC
https://trpc.io/docs/client/vanilla/infer-types

[11] [12] [13] [14] [15] [16] Mastering TypeScript Utility Types: Omit, Pick, and Partial for Cleaner Code | by Nata Nael | Jun, 2025 | Stackademic
https://blog.stackademic.com/mastering-typescript-utility-types-omit-pick-and-partial-for-cleaner-code-c28469f1bfcf?gi=41396c0a87f8