TypeScript 高级技巧

追逐者
web前端一线搬运工

关注他

409 人赞同了该文章

用了一段时间的 typescript 之后，深感中大型项目中 typescript 的必要性，它能够提前在编译期避免许多 bug，如很恶心的拼写问题。而越来越多的 package 也开始使用 ts ，学习 ts 已是势在必行。

以下是我在工作中总结到的比较实用的 typescript 技巧。

01 keyof

keyof 与 Object.keys 略有相似，只不过 keyof 取 interface 的键。

```
interface Point {
    x: number;
    y: number;
}

// type keys = "x" | "y"
type keys = keyof Point;
```

假设有一个 object 如下所示，我们需要使用 typescript 实现一个 get 函数来获取它的属性值

```
const data = {
  a: 3,
  hello: 'world'
}

function get(o: object, name: string) {
  return o[name]
}
```

我们刚开始可能会这么写，不过它有很多缺点

1. 无法确认返回类型：这将损失 ts 最大的类型校验功能
2. 无法对 key 做约束：可能会犯拼写错误的问题

这时可以使用 keyof 来加强 get 函数的类型功能，有兴趣的同学可以看看 _.get 的 type 标记以及实现

```
function get<T extends object, K extends keyof T>(o: T, name: K): T[K] {
  return o[name]
}
```

02 Partial & Pick

既然了解了 keyof ，可以使用它对属性做一些扩展， 如实现 Partial 和 Pick ， Pick 一般用在 _.pick 中

```
type Partial<T> = {
  [P in keyof T]?: T[P];
};

type Pick<T, K extends keyof T> = {
  [P in K]: T[P];
};

interface User {
  id: number;
  age: number;
  name: string;
};

// 相当于: type PartialUser = { id?: number; age?: number; name?: string; }
type PartialUser = Partial<User>

// 相当于: type PickUser = { id: number; age: number; }
type PickUser = Pick<User, "id" | "age">
```

03 Condition Type

类似于 js 中的 ?: 运算符，可以使用它扩展一些基本类型

```
T extends U ? X : Y

type isTrue<T> = T extends true ? true : false
// 相当于 type t = false
```

```
type t = isTrue<number>

// 相当于 type t = false
type t1 = isTrue<false>
```

04 never & Exclude & Omit

官方文档对 never 的描述如下

> the never type represents the type of values that never occur.

结合 never 与 conditional type 可以推出很多有意思而且实用的类型，比如 Omit

```
type Exclude<T, U> = T extends U ? never : T;

// 相当于: type A = 'a'
type A = Exclude<'x' | 'a', 'x' | 'y' | 'z'>
```

结合 Exclude 可以推出 Omit 的写法

```
type Omit<T, K extends keyof any> = Pick<T, Exclude<keyof T, K>>;

interface User {
  id: number;
  age: number;
  name: string;
};

// 相当于: type PickUser = { age: number; name: string; }
type OmitUser = Omit<User, "id">
```

05 typeof

顾名思义， typeof 代表取某个值的 type，可以从以下示例来展示他们的用法

```
const a: number = 3

// 相当于: const b: number = 4
const b: typeof a = 4
```

在一个典型的服务端项目中，我们经常需要把一些工具塞到 context 中，如config，logger，db models, utils 等，此时就使用到 typeof 。

```
import logger from './logger'
import utils from './utils'

interface Context extends KoaContect {
  logger: typeof logger,
  utils: typeof utils
}

app.use((ctx: Context) => {
  ctx.logger.info('hello, world')

  // 会报错，因为 logger.ts 中没有暴露此方法，可以最大限度的避免拼写错误
  ctx.loger.info('hello, world')
})
```

06 is

在此之前，先看一个 koa 的错误处理流程，以下是对 error 进行集中处理，并且标识 code 的过程

```
app.use(async (ctx, next) => {
  try {
    await next();
  } catch (err) {
    let code = 'BAD_REQUEST'
    if (err.isAxiosError) {
      code = `Axios-${err.code}`
    } else if (err instanceof Sequelize.BaseError) {

    }
    ctx.body = {
      code
    }
  }
})
```

在 err.code 处，会编译出错，提示 Property 'code' does not exist on type 'Error'.ts(2339) 。

此时可以使用 as AxiosError 或者 as any 来避免报错，不过强制类型转换也不够友好

```
if ((err as AxiosError).isAxiosError) {
  code = `Axios-${(err as AxiosError).code}`
}
```

此时可以使用 is 来判定值的类型

```
function isAxiosError (error: any): error is AxiosError {
  return error.isAxiosError
}

if (isAxiosError(err)) {
  code = `Axios-${err.code}`
}
```

在 GraphQL 的源码中，有很多诸如此类的用法，用以标识类型

```
export function isType(type: any): type is GraphQLType;

export function isScalarType(type: any): type is GraphQLScalarType;

export function isObjectType(type: any): type is GraphQLObjectType;

export function isInterfaceType(type: any): type is GraphQLInterfaceType;
```

07 interface & type

interface 与 type 的区别是什么？可以参考以下 stackoverflow 的问题

https://stackoverflow.com/que...

一般来说， interface 与 type 区别很小，比如以下两种写法差不多

```
interface A {
  a: number;
  b: number;
};

type B {
```

```
  a: number;
  b: number;
}
```

其中 interface 可以如下合并多个，而 type 只能使用 & 类进行连接。

```
interface A {
    a: number;
}

interface A {
    b: number;
}

const a: A = {
    a: 3,
    b: 4
}
```

08 Dictionary & Many

这几个语法糖是从 lodash 的 types 源码中学到的，平时工作中的使用频率还挺高。

```
interface Dictionary<T> {
  [index: string]: T;
};

interface NumericDictionary<T> {
  [index: number]: T;
};

const data:Dictionary<number> = {
  a: 3,
  b: 4
}
```

09 使用 const enum 维护常量表

相比使用字面量对象维护常量， const enum 可以提供更安全的类型检查

```
// 使用 object 维护常量
const enum TODO_STATUS {
    TODO = 'TODO',
    DONE = 'DONE',
    DOING = 'DOING'
}

function todos (status:keyof typeof TODO_STATUS): string {
    return TODO_STATUS[status as keyof typeof TODO_STATUS]
}
```

10 VS Code Tips & Typescript Command

使用 VS Code 有时会出现，使用 `tsc` 编译时产生的问题与 `vs code` 提示的问题不一致

找到项目右下角的 `Typescript` 字样，右侧显示它的版本号，可以点击选择 `Use Workspace Version` ，它表示与项目依赖的 typescript 版本一致。

或者编辑 `.vs-code/settings.json`

```
{
  "typescript.tsdk": "node_modules/typescript/lib"
}
```

11 PropertyKey

type PropertyKey=string|number|symbol

```
const a = new Map<PropertyKey, any>();
a.set(1, 'test')
a.set('1', 'test')
a.set(Symbol('symbol'), 'test')
```

12 Typescript Roadmap

最后一条也是最重要的一条，翻阅 `Roadmap` ，了解 `ts` 的一些新的特性与 bug 修复情况。

编辑于 05-01

「真诚赞赏，手留余香」

赞赏

还没有人赞赏，快来当第一个赞赏的人吧！

前端开发 TypeScript JavaScript