
TypeScript类型编程

类型验证的好处

如何保证`add`函数是接受的参数都是`number`,

```
function add(x, y) {
  return x + y;
}
```

在运行时的时候手动验证

```
function add(x, y) {
  if(typeof x === 'number' && typeof y === 'number') {
    return x + y;
  }
  throw error;
}
```

利用第三方工具验证

```
import {object, number} from "superstruct"
function add(x, y){
  const struct = object({
    x: number(),
    y: number()
  })
  assert(struct, {x, y});
  return x + y;
}
```

利用ts在编译时的时候验证

```
function add(x:number, y:number) {
  return x + y;
}

add('1', 1);
```

类型验证	验证时间	优点	缺点
js	无	项目规模小时候配合灵活，开发速度块	维护困难
ts	编译时	编译时验证，需要用户书写各种类型,配合编辑器在开发时就能避免一些错误	无
第三方工具	运行时	对于排查错误(甩锅)，项目代码健壮性有作用	引入额外的依赖

值空间和类型空间

值

> 字面量(literal)、变量(variable)、常量(constant)、函数形参、函数对象、`class`、`enum`...

类型

> `string`,`number`,`function`,`type` 关键字定义的类型 `interface`、`class` 和 `enum`...

TS类型检查的限制-类型擦除

TS最终是会编译到JavaScript，TS的类型会被擦除

> TypeScript 3.8 引入了`import type`的语法，可以显式指明引入类型，从而杜绝值空间潜在的循环引用问题

类型推断

自动类型推断

```
function add<T extends number>(x: T, y: T){
  return x + y;
}

const res1 = add<number>(1,2); // number
const res2 = add('1',2); // type check error
const res3 = add<string>('1','2'); // typecheckerror
```

前置知识

- 联合类型
- 交叉类型
- 映射类型
- 索引类型
  - `keyof T` 获取所有属性名联合
  - `T[K]` 获取某个属性的类型
- `extends ?` 分支
- 递归 循环

类型的基本操作

- 联合类型

```
  type Union = string | number;
```

```
type Dog = {
  walk: () => {}
  bark: () => {}
}

type Duck = {
  walk: () => {}
}

function walk(animal: Dog | Duck) {
  animal.walk();
}
```

而oop的做法会去设计一个抽象类，里面包含walk方法，然后再实现一个狗类和一个鸭子类，然后调用这个方法

交叉类型是将多个类型合并为一个类型。 这让我们可以把现有的多种类型叠加到一起成为一种类型，它包含了所需的所有类型的特性。 例如下面这个例子，这两个`Vip`和`Visitor`是有两个

- 交叉类型

```
type ErrorHandler = {
  success: boolean
  error: Error
}

type Vip = {
  desc: string
}
type Visitor = {
  desc: string
}

type PersonVipResponse = ErrorHandler & Vip
type PersonVisitorResponse = ErrorHandler & Visitor
```

递归

TypeScript 允许递归构造类型，而递归和循环是等价的,下面这个List工具类型会构造出定长的元组

```
// 返回一个定长的元组类型
type List<Type, n extends number, result extends any[] = []> =
    result['length'] extends n ? result :
      List<Type, n, [...result, Type]>;


type tuple_3 = List<number, 3>; // [number, number, number]
```

`extends ?` 分支

extends 关键字给我们提供了类似`if else`分支的能力,能根据输入的类型决定输出的类型

```
type result<T> = T extends number ? true : false;

type res_number = result<0>
type res_string = result<0>
```

需要注意的是联合类型使用extends时 这里的结果`string[] | number[]`而不是`Array<string|number>`

```
type ToArray<T> =  T extends any ? T[] : never;

type res = ToArray<string | number>; // string[] | number[]
```

索引类型

可以使用索引类型来查询另一个类型上某个属性的类型

```
type Person = {
  name: string
  id: number
}

type filter = Person['name'|'id'];
```

映射类型

映射类型有点类似js里面的`object.keys()`,通过`keyof`能获取到所有键的联合类型用来遍历键

```
type OptionsFlags<Type> = {
  [Property in keyof Type]: boolean;
};

type FeatureFlags = {
  darkMode: () => void;
  newUserProfile: () => void;
};

type FeatureOptions = OptionsFlags<FeatureFlags>;
```

一个典型的工具类型`Awaited`的实现也是递归的

```
type _Awaited<T> =
    T extends null | undefined ? T : // special case for `null | undefined` when not in `--strictNullChecks` mode
        T extends object & { then(onfulfilled: infer F): any } ? // `await` only unwraps object types with a callable `then
            F extends ((value: infer V, ...args: any) => any) ? // if the argument to `then` is callable, extracts the firs
                _Awaited<V> : // recursively unwrap the value
                never : // the argument to `then` was not callable
        T; // non-object or non-thenable
type result = _Awaited<Promise<Promise<number>>>; // string
```

一些好用的工具类型

- omit (忽略type当中的某个属性)
- pick (选择type当中的某些属性)
- exclude (排除联合类型中某些类型)
- ReturnType
- Awaited
- …

针对类型的编程

利用typescript的类型系统，实现一个获取第n项*Fibonacci*数列的值 [1、 1、 2、 3、 5、 8、 13、 21、 34、 55、 89、 144、 233、 377、 610、 987.....]

```
type List<Type, n extends number, result extends any[] = []> =
    result['length'] extends n ? result :
      List<Type, n, [...result, Type]>;

// 利用List实现加法
type length<l extends any[] = []> = l['length'] extends number ? l['length'] : 0;
type add<x extends number, y extends number> = length<[...List<number, x>, ...List<number,y>]>;
// type minus<x,y> =

type fib<n extends number, x extends number = 1, y extends number = 1, counter extends number = 1> = counter extends n ? x
  fib<n, y, add<x, y>, add<counter, 1>>

type res_7 = fib<7>
type res_8 = fib<8>
type res_9 = fib<9>
type res_10 = fib<10>


// 更多操作 参考 https://zhuanlan.zhihu.com/p/426966480
```

reference

- 读懂类型体操：TypeScript 类型元编程基础入门
- typescript handlebook
- TypeScript类型元编程入门指南