TypeScript 类型工具

TypeScript 提供了一些内置的类型工具，用来方便地处理各种类型，以及生成新的类型。

TypeScript 内置了 17 个类型工具，可以直接使用。

Awaited<Type>

`Awaited<Type>` 用来取出 Promise 的返回值类型，适合用在描述 `then()` 方法和 await 命令的参数类型。

typescript

```
// string
type A = Awaited<Promise<string>>;
```

上面示例中， `Awaited<Type>` 会返回 Promise 的返回值类型（string）。

它也可以返回多重 Promise 的返回值类型。

typescript

```
// number
type B = Awaited<Promise<Promise<number>>>;
```

如果它的类型参数不是 Promise 类型，那么就会原样返回。

typescript

```
// number | boolean
type C = Awaited<boolean | Promise<number>>;
```

上面示例中，类型参数是一个联合类型，其中的 `boolean` 会原样返回，所以最终返回的是 `number|boolean` 。

`Awaited<Type>` 的实现如下。

typescript

```typescript
type Awaited<T> =
  T extends null | undefined ? T :
  T extends object & {
    then(
      onfulfilled: infer F,
      ...args: infer _
    ): any;
  } ? F extends (
    value: infer V,
    ...args: infer _
  ) => any ? Awaited<...> : never:
  T;
```

ConstructorParameters<Type>

`ConstructorParameters<Type>` 提取构造方法 `Type` 的参数类型，组成一个元组类型返回。

typescript

```typescript
type T1 = ConstructorParameters<new (x: string, y: number) => object>; // [x: string

type T2 = ConstructorParameters<new (x?: string) => object>; // [x?: string | undefi
```

它可以返回一些内置构造方法的参数类型。

typescript

```typescript
type T1 = ConstructorParameters<ErrorConstructor>; // [message?: string]

type T2 = ConstructorParameters<FunctionConstructor>; // string[]

type T3 = ConstructorParameters<RegExpConstructor>; // [pattern:string|RegExp, flags
```

如果参数类型不是构造方法，就会报错。

typescript

```typescript
type T1 = ConstructorParameters<string>; // 报错

type T2 = ConstructorParameters<Function>; // 报错
```

any 类型和 never 类型是两个特殊值，分别返回 unknown[] 和 never 。

typescript

```typescript
type T1 = ConstructorParameters<any>; // unknown[]

type T2 = ConstructorParameters<never>; // never
```

ConstructorParameters<Type> 的实现如下。

typescript

```typescript
type ConstructorParameters<T extends abstract new (...args: any) => any> =
  T extends abstract new (...args: infer P) => any ? P : never;
```

Exclude<UnionType, ExcludedMembers>

Exclude<UnionType, ExcludedMembers> 用来从联合类型 UnionType 里面，删除某些类型 ExcludedMembers ，组成一个新的类型返回。

typescript

```typescript
type T1 = Exclude<"a" | "b" | "c", "a">; // 'b'|'c'
type T2 = Exclude<"a" | "b" | "c", "a" | "b">; // 'c'
type T3 = Exclude<string | (() => void), Function>; // string
type T4 = Exclude<string | string[], any[]>; // string
type T5 = Exclude<(() => void) | null, Function>; // null
type T6 = Exclude<200 | 400, 200 | 201>; // 400
type T7 = Exclude<number, boolean>; // number
```

Exclude<UnionType, ExcludedMembers> 的实现如下。

typescript

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

上面代码中，等号右边的部分，表示先判断 T 是否兼容 U ，如果是的就返回 never 类型，否则返回当前类型 T 。由于 never 类型是任何其他类型的子类型，它跟其他类型组成联合类型时，可以直接将 never 类型从联合类型中“消掉”，因此 Exclude<T, U> 就相当于删除兼容的类型，剩下不兼容的类型。

Extract<Type, Union>

`Extract<UnionType, Union>` 用来从联合类型 `UnionType` 之中，提取指定类型 `Union` ，组成一个新类型返回。它与 `Exclude<T, U>` 正好相反。

typescript

```
type T1 = Extract<"a" | "b" | "c", "a">; // 'a'
type T2 = Extract<"a" | "b" | "c", "a" | "b">; // 'a'|'b'
type T3 = Extract<"a" | "b" | "c", "a" | "d">; // 'a'
type T4 = Extract<string | string[], any[]>; // string[]
type T5 = Extract<(() => void) | null, Function>; // () => void
type T6 = Extract<200 | 400, 200 | 201>; // 200
```

如果参数类型 `Union` 不包含在联合类型 `UnionType` 之中，则返回 `never` 类型。

typescript

```
type T = Extract<string | number, boolean>; // never
```

`Extract<UnionType, Union>` 的实现如下。

typescript

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

InstanceType<Type>

`InstanceType<Type>` 提取构造函数的返回值的类型（即实例类型），参数 `Type` 是一个构造函数，等同于构造函数的 `ReturnType<Type>` 。

typescript

```
type T = InstanceType<new () => object>; // object
```

上面示例中，类型参数是一个构造函数 `new () => object` ，返回值是该构造函数的实例类型（ `object` ）。

下面是一些例子。

typescript

```
type A = InstanceType<ErrorConstructor>; // Error
type B = InstanceType<FunctionConstructor>; // Function
type C = InstanceType<RegExpConstructor>; // RegExp
```

上面示例中， `InstanceType<T>` 的参数都是 TypeScript 内置的原生对象的构造函数类型，`InstanceType<T>` 的返回值就是这些构造函数的实例类型。

由于 Class 作为类型，代表实例类型。要获取它的构造方法，必须把它当成值，然后用 `typeof` 运算符获取它的构造方法类型。

typescript

```
class C {
  x = 0;
  y = 0;
}

type T = InstanceType<typeof C>; // C
```

上面示例中， `typeof C` 是 `C` 的构造方法类型，然后 InstanceType 就能获得实例类型，即 `C` 本身。

如果类型参数不是构造方法，就会报错。

typescript

```
type T1 = InstanceType<string>; // 报错

type T2 = InstanceType<Function>; // 报错
```

如果类型参数是 `any` 或 `never` 两个特殊值，分别返回 `any` 和 `never` 。

typescript

```
type T1 = InstanceType<any>; // any

type T2 = InstanceType<never>; // never
```

`InstanceType<Type>` 的实现如下。

typescript

```
type InstanceType<T extends abstract new (...args: any) => any> =
  T extends abstract new (...args: any) => infer R ? R : any;
```

NonNullable<Type>

`NonNullable<Type>` 用来从联合类型 `Type` 删除 `null` 类型和 `undefined` 类型，组成一个新类型返回，也就是返回 `Type` 的非空类型版本。

typescript

```typescript
// string|number
type T1 = NonNullable<string | number | undefined>;

// string[]
type T2 = NonNullable<string[] | null | undefined>;

type T3 = NonNullable<boolean>; // boolean
type T4 = NonNullable<number | null>; // number
type T5 = NonNullable<string | undefined>; // string
type T6 = NonNullable<null | undefined>; // never
```

`NonNullable<Type>` 的实现如下。

typescript

```typescript
type NonNullable<T> = T & {};
```

上面代码中， `T & {}` 等同于求 `T & Object` 的交叉类型。由于 TypeScript 的非空值都属于 `Object` 的子类型，所以会返回自身；而 `null` 和 `undefined` 不属于 `Object` ，会返回 `never` 类型。

Omit<Type, Keys>

`Omit<Type, Keys>` 用来从对象类型 `Type` 中，删除指定的属性 `Keys` ，组成一个新的对象类型返回。

typescript

```typescript
interface A {
  x: number;
  y: number;
}

type T1 = Omit<A, "x">; // { y: number }
```

```
type T2 = Omit<A, "y">; // { x: number }
type T3 = Omit<A, "x" | "y">; // { }
```

上面示例中， `Omit<Type, Keys>` 从对象类型 `A` 里面删除指定属性，返回剩下的属性。

指定删除的键名 `Keys` 可以是对象类型 `Type` 中不存在的属性，但必须兼容 `string|number|symbol` 。

typescript

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

type T = Omit<A, "z">; // { x: number; y: number }
```

上面示例中，对象类型 `A` 中不存在属性 `z` ，所以就原样返回了。

`Omit<Type, Keys>` 的实现如下。

typescript

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

OmitThisParameter<Type>

`OmitThisParameter<Type>` 从函数类型中移除 this 参数。

typescript

```
function toHex(this: Number) {
  return this.toString(16);
}

type T = OmitThisParameter<typeof toHex>; // () => string
```

上面示例中， `OmitThisParameter<T>` 给出了函数 `toHex()` 的类型，并将其中的 `this` 参数删除。

如果函数没有 this 参数，则返回原始函数类型。

`OmitThisParameter<Type>` 的实现如下。

```typescript
type OmitThisParameter<T> = unknown extends ThisParameterType<T>
  ? T
  : T extends (...args: infer A) => infer R
  ? (...args: A) => R
  : T;
```

Parameters<Type>

`Parameters<Type>` 从函数类型 `Type` 里面提取参数类型，组成一个元组返回。

```typescript
type T1 = Parameters<() => string>; // []

type T2 = Parameters<(s: string) => void>; // [s:string]

type T3 = Parameters<<T>(arg: T) => T>; // [arg: unknown]

type T4 = Parameters<(x: { a: number; b: string }) => void>; // [x: { a: number, b:

type T5 = Parameters<(a: number, b: number) => number>; // [a:number, b:number]
```

上面示例中， `Parameters<Type>` 的返回值会包括函数的参数名，这一点需要注意。

如果参数类型 `Type` 不是带有参数的函数形式，会报错。

```typescript
// 报错
type T1 = Parameters<string>;

// 报错
type T2 = Parameters<Function>;
```

由于 `any` 和 `never` 是两个特殊值，会返回 `unknown[]` 和 `never` 。

```typescript
type T1 = Parameters<any>; // unknown[]
```

```typescript
type T2 = Parameters<never>; // never
```

`Parameters<Type>` 主要用于从外部模块提供的函数类型中，获取参数类型。

typescript

```typescript
interface SecretName {
  first: string;
  last: string;
}

interface SecretSanta {
  name: SecretName;
  gift: string;
}

export function getGift(name: SecretName, gift: string): SecretSanta {
  // ...
}
```

上面示例中，模块只输出了函数 `getGift()` ，没有输出参数 `SecretName` 和返回值 `SecretSanta` 。这时就可以通过 `Parameters<T>` 和 `ReturnType<T>` 拿到这两个接口类型。

typescript

```typescript
type ParaT = Parameters<typeof getGift>[0]; // SecretName

type ReturnT = ReturnType<typeof getGift>; // SecretSanta
```

`Parameters<Type>` 的实现如下。

typescript

```typescript
type Parameters<T extends (...args: any) => any> = T extends (
  ...args: infer P
) => any
  ? P
  : never;
```

Partial<Type>

`Partial<Type>` 返回一个新类型，将参数类型 `Type` 的所有属性变为可选属性。

typescript

```typescript
interface A {
  x: number;
  y: number;
}

type T = Partial<A>; // { x?: number; y?: number; }
```

`Partial<Type>` 的实现如下。

typescript

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

Pick<Type, Keys>

`Pick<Type, Keys>` 返回一个新的对象类型，第一个参数 `Type` 是一个对象类型，第二个参数 `Keys` 是 `Type` 里面被选定的键名。

typescript

```typescript
interface A {
  x: number;
  y: number;
}

type T1 = Pick<A, "x">; // { x: number }
type T2 = Pick<A, "y">; // { y: number }
type T3 = Pick<A, "x" | "y">; // { x: number; y: number }
```

上面示例中， `Pick<Type, Keys>` 会从对象类型 `A` 里面挑出指定的键名，组成一个新的对象类型。

指定的键名 `Keys` 必须是对象键名 `Type` 里面已经存在的键名，否则会报错。

typescript

```typescript
interface A {
  x: number;
  y: number;
}
```

```
type T = Pick<A, "z">; // 报错
```

上面示例中，对象类型 `A` 不存在键名 `z` ，所以报错了。

`Pick<Type, Keys>` 的实现如下。

typescript

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

Readonly<Type>

`Readonly<Type>` 返回一个新类型，将参数类型 `Type` 的所有属性变为只读属性。

typescript

```
interface A {
  x: number;
  y?: number;
}

// { readonly x: number; readonly y?: number; }
type T = Readonly<A>;
```

上面示例中， `y` 是可选属性， `Readonly<Type>` 不会改变这一点，只会让 `y` 变成只读。

`Readonly<Type>` 的实现如下。

typescript

```
type Readonly<T> = {
  readonly [P in keyof T]: T[P];
};
```

我们可以自定义类型工具 `Mutable<Type>` ，将参数类型的所有属性变成可变属性。

typescript

```
type Mutable<T> = {
  -readonly [P in keyof T]: T[P];
```

```
};
```

上面代码中， `-readonly` 表示去除属性的只读标志。

相应地， `+readonly` 就表示增加只读标志，等同于 `readonly` 。因此， `ReadOnly<Type>` 的实现也可以写成下面这样。

typescript

```
type Readonly<T> = {
  +readonly [P in keyof T]: T[P];
};
```

`Readonly<Type>` 可以与 `Partial<Type>` 结合使用，将所有属性变成只读的可选属性。

typescript

```
interface Person {
  name: string;
  age: number;
}

const worker: Readonly<Partial<Person>> = { name: "张三" };

worker.name = "李四"; // 报错
```

Record<Keys, Type>

`Record<Keys, Type>` 返回一个对象类型，参数 `Keys` 用作键名，参数 `Type` 用作键值类型。

typescript

```
// { a: number }
type T = Record<"a", number>;
```

上面示例中， `Record<Keys, Type>` 的第一个参数 `a` ，用作对象的键名，第二个参数 `number` 是 `a` 的键值类型。

参数 `Keys` 可以是联合类型，这时会依次展开为多个键。

typescript

```
// { a: number, b: number }
```

```typescript
type T = Record<"a" | "b", number>;
```

上面示例中，第一个参数是联合类型 `'a'|'b'` ，展开成两个键名 `a` 和 `b` 。

如果参数 `Type` 是联合类型，就表明键值是联合类型。

```typescript
// { a: number|string }
type T = Record<"a", number | string>;
```

参数 `Keys` 的类型必须兼容 `string|number|symbol` ，否则不能用作键名，会报错。

`Record<Keys, Type>` 的实现如下。

```typescript
type Record<K extends string | number | symbol, T> = { [P in K]: T };
```

Required<Type>

`Required<Type>` 返回一个新类型，将参数类型 `Type` 的所有属性变为必选属性。它与 `Partial<Type>` 的作用正好相反。

```typescript
interface A {
  x?: number;
  y: number;
}

type T = Required<A>; // { x: number; y: number; }
```

`Required<Type>` 的实现如下。

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

上面代码中，符号 `-?` 表示去除可选属性的“问号”，使其变成必选属性。

相对应地，符号 `+?` 表示增加可选属性的“问号”，等同于 `?` 。因此，前面的 `Partial<Type>` 的定义也可以写成下面这样。

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

ReadonlyArray<Type>

`ReadonlyArray<Type>` 用来生成一个只读数组类型，类型参数 `Type` 表示数组成员的类型。

```typescript
const values: ReadonlyArray<string> = ["a", "b", "c"];

values[0] = "x"; // 报错
values.push("x"); // 报错
values.pop(); // 报错
values.splice(1, 1); // 报错
```

上面示例中，变量 `values` 的类型是一个只读数组，所以修改成员会报错，并且那些会修改源数组的方法 `push()` 、 `pop()` 、 `splice()` 等都不存在。

`ReadonlyArray<Type>` 的实现如下。

```typescript
interface ReadonlyArray<T> {
  readonly length: number;

  readonly [n: number]: T;

  // ...
}
```

ReturnType<Type>

`ReturnType<Type>` 提取函数类型 `Type` 的返回值类型，作为一个新类型返回。

typescript

```typescript
type T1 = ReturnType<() => string>; // string

type T2 = ReturnType<
  () => {
    a: string;
    b: number;
  }
>; // { a: string; b: number }

type T3 = ReturnType<(s: string) => void>; // void

type T4 = ReturnType<() => () => any[]>; // () => any[]

type T5 = ReturnType<typeof Math.random>; // number

type T6 = ReturnType<typeof Array.isArray>; // boolean
```

如果参数类型是泛型函数，返回值取决于泛型类型。如果泛型不带有限制条件，就会返回 `unknown` 。

typescript

```typescript
type T1 = ReturnType<<T>() => T>; // unknown

type T2 = ReturnType<<T extends U, U extends number[]>() => T>; // number[]
```

如果类型不是函数，会报错。

typescript

```typescript
type T1 = ReturnType<boolean>; // 报错

type T2 = ReturnType<Function>; // 报错
```

`any` 和 `never` 是两个特殊值，分别返回 `any` 和 `never` 。

typescript

```typescript
type T1 = ReturnType<any>; // any

type T2 = ReturnType<never>; // never
```

`ReturnType<Type>` 的实现如下。

typescript

```typescript
type ReturnType<T extends (...args: any) => any> = T extends (
  ...args: any
) => infer R
  ? R
  : any;
```

ThisParameterType<Type>

`ThisParameterType<Type>` 提取函数类型中 `this` 参数的类型。

typescript

```typescript
function toHex(this: Number) {
  return this.toString(16);
}

type T = ThisParameterType<typeof toHex>; // number
```

如果函数没有 `this` 参数，则返回 `unknown` 。

`ThisParameterType<Type>` 的实现如下。

typescript

```typescript
type ThisParameterType<T> =
  T extends (
    this: infer U,
    ...args: never
  ) => any ? U : unknown;
```

ThisType<Type>

`ThisType<Type>` 不返回类型，只用来跟其他类型组成交叉类型，用来提示 TypeScript 其他类型里面的 `this` 的类型。

typescript

```typescript
interface HelperThisValue {
  logError: (error: string) => void;
}
```

```
let helperFunctions: { [name: string]: Function } & ThisType<HelperThisValue> =
  {
    hello: function () {
      this.logError("Error: Something wrong!"); // 正确
      this.update(); // 报错
    },
  };
```

上面示例中，变量 `helperFunctions` 的类型是一个正常的对象类型与 `ThisType<HelperThisValue>` 组成的交叉类型。

这里的 `ThisType` 的作用是提示 TypeScript，变量 `helperFunctions` 的 `this` 应该满足 `HelperThisValue` 的条件。所以， `this.logError()` 可以正确调用，而 `this.update()` 会报错，因为 `HelperThisValue` 里面没有这个方法。

注意，使用这个类型工具时，必须打开 `noImplicitThis` 设置。

下面是另一个例子。

typescript

```
let obj: ThisType<{ x: number }> & { getX: () => number };

obj = {
  getX() {
    return this.x + this.y; // 报错
  },
};
```

上面示例中， `getX()` 里面的 `this.y` 会报错，因为根据 `ThisType<{ x: number }>` ，这个对象的 `this` 不包含属性 `y` 。

`ThisType<Type>` 的实现就是一个空接口。

typescript

```
interface ThisType<T> {}
```

字符串类型工具

TypeScript 内置了四个字符串类型工具，专门用来操作字符串类型。这四个工具类型都定义在 TypeScript 自带的 `.d.ts` 文件里面。

它们的实现都是在底层调用 JavaScript 引擎提供 JavaScript 字符操作方法。

Uppercase<StringType>

`Uppercase<StringType>` 将字符串类型的每个字符转为大写。

typescript

```typescript
type A = "hello";

// "HELLO"
type B = Uppercase<A>;
```

上面示例中， `Uppercase<T>` 将 hello 转为 HELLO。

Lowercase<StringType>

`Lowercase<StringType>` 将字符串的每个字符转为小写。

typescript

```typescript
type A = "HELLO";

// "hello"
type B = Lowercase<A>;
```

上面示例中， `Lowercase<T>` 将 HELLO 转为 hello。

Capitalize<StringType>

`Capitalize<StringType>` 将字符串的第一个字符转为大写。

typescript

```typescript
type A = "hello";

// "Hello"
type B = Capitalize<A>;
```

上面示例中， `Capitalize<T>` 将 hello 转为 Hello。

Uncapitalize<StringType>

`Uncapitalize<StringType>` 将字符串的第一个字符转为小写。

```typescript
type A = "HELLO";

// "hELLO"
type B = Uncapitalize<A>;
```

上面示例中， `Uncapitalize<T>` 将 HELLO 转为 hELLO。

参考链接

- What is TypeScript's ThisType used for?

最后更新: 2023/8/13 15:25

Previous page
类型映射

Next page
注释指令