TypeScript 类型工具

目录
Awaited<Type>
ConstructorParameters<Type> Exclude<UnionType, ExcludedMembers> Extract<UnionType, Union>
InstanceType<Type>
NonNullable<Type>
Omit<Type, Keys>
OmitThisParameter<Type>
Parameters<Type>
Partial<Type>
Pick<Type, Keys>
Readonly<Type>
Record<Keys, Type>
Required<Type>
ReadonlyArray<Type>
ReturnType<Type>
ThisParameterType<Type>
ThisType<Type>
字符串类型工具
Uppercase<StringType>
Lowercase<StringType>
Capitalize<StringType>
Uncapitalize<StringType>
参考链接

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。

这些类型工具都是语言本身提供的，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。

```
// 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。

```
// 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。

```
// 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。

```
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<V> : 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。

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

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

□□□□□□□□□□□□□□□□□□□□

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

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

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

□□□□□□□□□□□□□□□□□□□□

```
type T1 = ConstructorParameters<string>; // □□

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

`any` □□□ `never` □□□□□□□□□□□□□□□ `unknown[]` □ `never` □

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

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

`ConstructorParameters<Type>` □□□□□□

```
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> UnionType ExcludedMembers

```
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>

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

T U never T never never “ ” Exclude<T, U>

Extract<UnionType, Union>

Extract<UnionType, Union> UnionType Union Exclude<T, U>

```
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
```

Union UnionType never

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

Extract<UnionType, Union>

```
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>`。

```
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。

```
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。

```
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。

```
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`。

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

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

`InstanceType<Type>` □□□□□□

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

NonNullable<Type>

`NonNullable<Type>` □□□□□□□ `Type` □□ `null` □□□ `undefined` □□□□□□□□□□□□□□□□□□ `Type` □□□□□□□□

```
// 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>` □□□□□□

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

□□□□□□ `T & {}` □□□□ `T & Object` □□□□□□□□ TypeScript □□□□□□□ `Object` □□□□□□□□□□□□□□□ `null` □ `undefined` □□□ `Object` □□□□ `never` □□□

Omit<Type, Keys>

`Omit<Type, Keys>` □□□□□□□ `Type` □□□□□□□□□ `Keys` □□□□□□□□□□□□□□□

```
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`。

```
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。

```
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。

```
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>` □□□□□□

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

Parameters<Type>

`Parameters<Type>` □□□□□ `Type` □□□□□□□□□□□□□□□□□□□

```
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: string }]

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

□□□□□ `Parameters<Type>` □□□□□□□□□□□□□□□□□□□□□□□

□□□□□ `Type` □□□□□□□□□□□□□□□□

```
// □□
type T1 = Parameters<string>;

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

□□ `any` □ `never` □□□□□□□□□□ `unknown[]` □ `never` □

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

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

`Parameters<Type>` □□□□□□□□□□□□□□□□□□□□□□□□□□□□

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

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

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

□□□□□□□□□□□□□□ `getGift()` □□□□□□□ `SecretName` □□□□ `SecretSanta` □□□□□□□□ `Parameters<T>` □ `ReturnType<T>` □□□□□□□□□□

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

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

`Parameters<Type>` □□□□□□

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

`Partial<Type>`

`Partial<Type>` □□□□□□□□□□□□□□ `Type` □□□□□□□□□□□□

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

```
}

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

`Partial<Type>`的实现如下。

```
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。

```
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`必须是对象类型`Type`里面已经存在的键名，否则会报错。

```
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。

```
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。

```
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。

```
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。

```
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。

```
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。

```
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。

```
// { 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。

```
// { a: number, b: number }
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。

```
// { 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。

```
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。

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

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

`Required<Type>` 的实现如下。

```
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。

```
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。

```
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。

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

  readonly [n: number]: T;

  // ...
}
```

ReturnType<Type>

`ReturnType<Type>` □□□□□□ `Type` □□□□□□□□□□□□□□□□□

```
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
```

□□ `unknown` □

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

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

□□□□□□□□□□□□□

```
type T1 = ReturnType<boolean>; // □□

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

any □ never □□□□□□□□□□□□ any □ never □

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

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

ReturnType<Type> □□□□□□

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

ThisParameterType<Type>

ThisParameterType<Type> □□□□□□□ this □□□□□□

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

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

□□□□□□ this □□□□□□ unknown □

ThisParameterType<Type> □□□□□□

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

ThisType<Type>

ThisType<Type> □□□□□□□□□□□□□□□□□□□□□□□□□□□□ TypeScript □□□□□□□ this □□□□

```
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。

```
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。

```
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。

```
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。

```
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。

```
type A = 'hello';

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

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

Uncapitalize<StringType>

`Uncapitalize<StringType>` □□□□□□□□□□□□□□□

```
type A = 'HELLO';


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

□□□□□□ `Uncapitalize<T>`□ HELLO □□ hELLO□

□□□□

- What is TypeScript's ThisType used for?

□□□ #2
□ □□□ □□□ 2023-07-09 12:07:22 UTC
□ □□□ □□□ 2025-09-03 14:06:34 UTC