https://www.typescriptlang.org/docs/handbook/utility-types.html
https://www.typescriptlang.org/docs/handbook/utility-types.html

typescript 类型工具

Awaited<Type>

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

//type A = string

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

//type B = number

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

//type C = number | boolean
```

Partial<Type>

```
type A = {
  a: string;
  b: number;
  c: boolean;
};
type B = Partial<A>;

// type B={
//     a?:string;
//     b?:number;
//     c?:boolean;
// }
```

Required<Type>

```ts
type A = {
  a?: string;
  b?: number;
  c?: boolean;
};
type B = Required<A>;
// type B={
//     a:string;
//     b:number;
//     c:boolean;
// }
```

Readonly<Type>

```ts
type A = {
  a: string;
  b?: number;
  c: boolean;
};
type B = Readonly<A>;

// type A={
//   readonly a:string;
//   readonly b?:number;
//   readonly c:boolean;
// }
```

Record<Keys, Type>

```ts
type A = {
  a: string;
  b: number;
  c: boolean;
};
type B = 'AAA' | 'BBB' | 'CCC';
type C = Record<B, A>;

// type C={
//     AAA:{
//     a:string;
//     b:number;
//     c:boolean;
// },
// BBB:{
//     a:string;
//     b:number;
//     c:boolean;
// },
// CCC:{
```

```
//     a:string;
//     b:number;
//     c:boolean;
// }
// }
```

Pick<Type, Keys>

```
type A = {
  a: string;
  b: number;
  c: boolean;
};
type B = 'a' | 'b';
type C = Pick<A, B>;

// type C={
//      a:string;
//     b:number;
// }
```

Omit<Type, Keys>

```
type A = {
  a: string;
  b: number;
  c: boolean;
};
type B = 'a' | 'b';
type C = Omit<A, B>;

// type C={
//      c:boolean;
// }
```

Exclude<UnionType, ExcludedMembers>

```
type A = Exclude<1 | 2 | 3, 1>;
// type A=2|3;

type B = Exclude<string | number | (() => void), Function>;

//type B = string | number

type Shape =
  | { kind: 'circle'; radius: number }
  | { kind: 'square'; x: number }
  | { kind: 'triangle'; x: number; y: number };
```

```ts
type C = Exclude<Shape, { kind: 'circle' }>;
//type C ={ kind: "square"; x: number }  |{ kind: "triangle"; x: number; y:
number };
```

Extract<Type, Union>

```ts
type T0 = Extract<'a' | 'b' | 'c', 'a' | 'f'>;

// type T0 = "a"

type T1 = Extract<string | number | (() => void), Function>;

// type T1 = () => void

type Shape =
  | { kind: 'circle'; radius: number }
  | { kind: 'square'; x: number }
  | { kind: 'triangle'; x: number; y: number };

type T2 = Extract<Shape, { kind: 'circle' }>;

// type T2 = {
//     kind: "circle";
//     radius: number;
// }
```

NonNullable<Type>

```ts
type T0 = NonNullable<string | number | undefined>;

//type T0 = string | number

type T1 = NonNullable<string[] | null | undefined>;

//type T1 = string[]
```

Parameters<Type>

```ts
declare function f1(arg: { a: number; b: string }): void;

type T0 = Parameters<() => string>;

//type T0 = []

type T1 = Parameters<(s: string) => void>;

//type T1 = [s: string]
```

```ts
type T2 = Parameters<<T>(arg: T) => T>;

//type T2 = [arg: unknown]

type T3 = Parameters<typeof f1>;

// type T3 = [arg: {
//     a: number;
//     b: string;
// }]

type T4 = Parameters<any>;

//type T4 = unknown[]

type T5 = Parameters<never>;

//type T5 = never
```

ConstructorParameters<Type>

```ts
type T0 = ConstructorParameters<ErrorConstructor>;

//type T0 = [message?: string]

type T1 = ConstructorParameters<FunctionConstructor>;

//type T1 = string[]

type T2 = ConstructorParameters<RegExpConstructor>;

//type T2 = [pattern: string | RegExp, flags?: string]

class C {
  constructor(a: number, b: string) {}
}
type T3 = ConstructorParameters<typeof C>;

//type T3 = [a: number, b: string]

type T4 = ConstructorParameters<any>;

//type T4 = unknown[]
```

ReturnType<Type>

```
declare function f1(): { a: number; b: string };

type T0 = ReturnType<() => string>;

//type T0 = string

type T1 = ReturnType<(s: string) => void>;

//type T1 = void

type T2 = ReturnType<<T>() => T>;

//type T2 = unknown

type T3 = ReturnType<<T extends U, U extends number[]>() => T>;

//type T3 = number[]

type T4 = ReturnType<typeof f1>;

// type T4 = {
//     a: number;
//     b: string;
// }

type T5 = ReturnType<any>;

//type T5 = any

type T6 = ReturnType<never>;

//type T6 = never
```

InstanceType<Type>

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

type T0 = InstanceType<typeof C>;

//type T0 = C

type T1 = InstanceType<any>;

//type T1 = any
```

```
type T2 = InstanceType<never>;

//type T2 = never
```

NoInfer<Type>

```
function createStreetLight<C extends string>(colors: C[], defaultColor?:
NoInfer<C>) {
  // ...
}
createStreetLight(['red', 'yellow', 'green'], 'red'); // OK
createStreetLight(['red', 'yellow', 'green'], 'blue'); // Error
```

ThisParameterType<Type>

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

function numberToString(n: ThisParameterType<typeof toHex>) {
  return toHex.apply(n);
}
```

OmitThisParameter<Type>

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

const fiveToHex: OmitThisParameter<typeof toHex> = toHex.bind(5);

console.log(fiveToHex());
```

ThisType<Type>

```
type ObjectDescriptor<D, M> = {
  data?: D;
  methods?: M & ThisType<D & M>; // Type of 'this' in methods is D & M
};

function makeObject<D, M>(desc: ObjectDescriptor<D, M>): D & M {
  let data: object = desc.data || {};
  let methods: object = desc.methods || {};
  return { ...data, ...methods } as D & M;
}

let obj = makeObject({
```

```
  data: { x: 0, y: 0 },
  methods: {
    moveBy(dx: number, dy: number) {
      this.x += dx; // Strongly typed this
      this.y += dy; // Strongly typed this
    }
  }
});

obj.x = 10;
obj.y = 20;
obj.moveBy(5, 5);
```

typescript 使用注意事项

any 与 unknown

```
//nknown跟any的相似之处，在于所有类型的值都可以分配给unknown类型。
let x: unknown;

x = true; // 正确
x = 42; // 正确
x = 'Hello World'; // 正确

//unknown类型的变量，不能直接赋值给其他类型的变量
let v: unknown = 123;

let v1: boolean = v; // 报错
let v2: number = v; // 报错

//不能直接调用unknown类型变量的方法和属性
let v1: unknown = { foo: 123 };
v1.foo; // 报错

let v2: unknown = 'hello';
v2.trim(); // 报错

let v3: unknown = (n = 0) => n + 1;
v3(); // 报错

//unknown类型变量能够进行的运算是有限的，只能进行比较运算（运算符==、
===、!=、!==、||、&&、?）、取反运算（运算符!）、typeof运算符和instanceof运算符这几
种，其他运算都会报错。
let a: unknown = 1;

a + 1; // 报错
a === 1; // 正确

//unknown是严格版any
```

```ts
let s: unknown = 'hello';

if (typeof s === 'string') {
  s.length; // 正确
}
```

never 空类型

```ts
function fn(x: string | number) {
  if (typeof x === 'string') {
    // ...
  } else if (typeof x === 'number') {
    // ...
  } else {
    x; // never 类型
  }
}

//可以赋值给任意其他类型。
function f(): never {
  throw new Error('Error');
}
let v1: number = f(); // 不报错
let v2: string = f(); // 不报错
let v3: boolean = f(); // 不报错
```

typeof 运算符

```ts
typeof undefined; // "undefined"
typeof true; // "boolean"
typeof 1337; // "number"
typeof "foo"; // "string"
typeof {}; // "object"
typeof parseInt; // "function"
typeof Symbol(); // "symbol"
typeof 127n // "bigint"

let a=1;
type A=typeof a;//number
//type A=number

//typeof 的参数只能是标识符，不能是需要运算的表达式
type T = typeof Date(); // 报错

//typeof命令的参数不能是类型
type B=typeof A;//报错
```

type 交并

```
type Shape = {
  x: number;
  y: number;
} & ({ type: 'circle'; radius: number } | { type: 'rect'; width: number;
height: number });
```

数组只读

```
const arr: readonly number[] = [0, 1];

arr[1] = 2; // 报错
arr.push(3); // 报错
delete arr[0]; // 报错

//readonly关键字不能与数组的泛型写法一起使用。
// 报错
const arr: readonly Array<number> = [0, 1];

//TypeScript 提供了两个专门的泛型，用来生成只读数组的类型
const a1: ReadonlyArray<number> = [0, 1];

const a2: Readonly<number[]> = [0, 1];

//const断言
const arr = [0, 1] as const;

arr[0] = [2]; // 报错
```

数组与元组

```
// 数组
let a: number[] = [1];

// 元组
let t: [number] = [1];

let a: [number, number?] = [1];

//扩展运算符（...）用在元组的任意位置都可以，它的后面只能是一个数组或元组。
type t1 = [string, number, ...boolean[]];
type t2 = [string, ...boolean[], number];
type t3 = [...boolean[], string, number];

type Color = [red: number, green: number, blue: number];

const c: Color = [255, 255, 255];
```

```
type Tuple = [string, number];
type Age = Tuple[1]; // number
type TupleEl = Tuple[number]; // string|number

//只读
// 写法一
type t = readonly [number, string];

// 写法二
type t = Readonly<[number, string]>;

//错误写法
const arr = [1, 2];

function add(x: number, y: number) {
  // ...
}

add(...arr); // 报错

//正确写法
const arr: [number, number] = [1, 2];

function add(x: number, y: number) {
  // ...
}

add(...arr); // 正确
```

函数

```
// 写法一
const hello = function (txt: string) {
  console.log('hello ' + txt);
};

// 写法二
const hello: (txt: string) => void = function (txt) {
  console.log('hello ' + txt);
};

function f(x: number) {
  console.log(x);
}

f.version = '1.0';

let foo: {
  (x: number): void;
  version: string;
} = f;
```

```
//Function 类型的函数可以接受任意数量的参数，每个参数的类型都是any，返回值的类型也是any，代表没有任何约束
function doSomething(f: Function) {
  return f(1, 2, 3);
}

function f(x?: number) {
  return x;
}
f(); // OK
f(10); // OK
f(undefined); // 正确

//参数解构
function f([x, y]: [number, number]) {
  // ...
}

function sum({ a, b, c }: { a: number; b: number; c: number }) {
  console.log(a + b + c);
}

//
// rest 参数为数组
function joinNumbers(...nums: number[]) {
  // ...
}

// rest 参数为元组
function f(...args: [boolean, number]) {
  // ...
}
```

keyof typeof

```
const parts = {
  a: 1,
  b: 2
};
for (let k in parts) {
  console.log(parts[k as keyof typeof parts]);
}
```