#is

is 用于定义变量属于某个类型

下面判断时将出现类型错误提示

```
const isString = (x: unknown): boolean => typeof x === 'string'

function hd(a: unknown) {
  if (isString(a)) {
                //error: 对象的类型为 "unknown"。
    a.toUpperCase()
  }
}
let a = 'ab'

hd(a)

```

现在重新定义 isString 函数，使用 is 来定义变量为某个类型。

- x is string 表示如果函数返回值为 true，则 x 为 string 类型

```
const isString = (x: unknown): x is string => typeof x === 'string'

function hd(a: unknown) {
  if (isString(a)) {
    a.toUpperCase()
  }
}
let a = 'ab'

hd(a)

```

#keyof

获取类、接口索引组成的联合类型

- keyof 可用于基本数据类型、any、class、interface、enum 等

任何类型都可以使用 keyof

```
type HOUDUNREN = keyof string

let xj: HOUDUNREN = 'match'

```

索引类型使用 keyof 时，获取索引名

```
type HOUDUNREN = keyof { name: string, age: number }

let xj: HOUDUNREN = 'name'

```

下面是获取对象的属性的函数类型定义

```
function getAttribute<T>(obj: T, key: keyof T): T[keyof T] {
  return obj[key]
}

const user = { name: '后盾人', age: 18 }
getAttribute(user, 'name')

```

我们也可以用泛型定义索引类型

```
function getAttribute<T, D extends keyof T>(obj: T, key: D): T[D] {
  return obj[key]
}

const user = { name: '后盾人', age: 18 }
getAttribute(user, 'name')

```

#typeof

使用 typeof 可获取变量的类型，下面是获取字符串变量的类型

```
let hd = '后盾人'
```

```
2
3 // type HOUDUNREN = string
4 type HOUDUNREN = typeof hd
5
```

下面使用 typeof 获取对象的

```
1 let hd = { name: '后盾人', age: 18 }
2
3 /**
4  type HOUDUNREN = {
5   name: string;
6   age: number;
7  */
8 type HOUDUNREN = typeof hd
9
```

keyof 与 typeof 结合定义获取对象属性的函数

```
1 function getAttribute(obj: object, key: string) {
2   return obj[key as keyof typeof obj]
3 }
4
5 const hd = { name: 'houdunren' }
6 getAttribute(hd, 'name')
7
```

#in

in 用于遍历接口或联合类型的属性

- K in keyof T 指 K 类型为 keyof T 获取的 T 类型索引组成的联合类型

```
1 type USER = { name: string, age: number }
2
3 type MEMBER<T> = {
4  [K in keyof T]: T[K]
5 }
6
7 const hd: MEMBER<USER & { address: string }> = {
```

```
        age: 10, name: '后盾人', address: '上海',
}

```

#extends

extends 在 TS 中拥有多个特性，下面我们来分别了解。

#类型继承

extends 实现类型的继承

```
type XIANGJUNDASHU = { name: string }

interface houdunren extends XIANGJUNDASHU {
        age: number
}

const hd: houdunren = { age: 33, name: '后盾人' }

```

extends 可用于泛型的类型限定，下例中 T 必须包含 id、render 属性，即 T 类型可赋予 extends 右侧类型

```
function houdunren<T extends { id: number; render(n: number): number }>(arr: T[]) {
        arr.map((a) => a.render(a.id))
}

houdunren([{ id: 1, render(n) { return n } }])

```

#类型条件判断

extends 用于条件判断来决定返回什么类型，`A extends B ? true:false`。如果 A（狭窄类型） 可以赋予 B（宽泛类型） 类型则为 true。

- 下例的 hd 变量值必须为 false，因为 HOUDUNREN 不包含 XIANGJUNDASHU 类型

```
type XIANGJUNDASHU = { name: string, age: number }
```

```ts

type HOUDUNREN = { name: string }

type HDCMS = HOUDUNREN extends XIANGJUNDASHU ? true : false

const hd: HDCMS = false

```

下面是联合类型的条件判断

```ts
type XIANGJUNDASHU = string

type HOUDUNREN = string | number

const hd: HOUDUNREN extends XIANGJUNDASHU ? string : boolean = false //boolean

const xj: XIANGJUNDASHU extends HOUDUNREN ? string : boolean = '后盾人' //string

```

根据联合类型过滤掉指定索引

```ts
type User = { name: string, age: number, get(): void };

type FilterObjectProperty<T, U> = {
  [K in keyof T as Exclude<K, U>]: T[K]
}

type HD = FilterObjectProperty<User, 'name' | 'age'>

```

过滤掉指定的类型，以下代码含有下面几个含义

- 根据类型获取索引组合成的联合类型
- 根据新的联合类型提取出指定的索引，组合成新的类型

```ts
type USER = { name: string, age: number, get(a: string): void }

type FilterProperty<T, U> = {
  [K in keyof T]: T[K] extends U ? never : K
}[keyof T]
```

```
6
7 type UserType = Pick<USER, FilterProperty<USER, Function | number>>
8
```

#泛型条件分配

如果泛型是普通类型，则与上面一样也是判断左侧类型是否可赋予右侧类型

```
1 type XIANGJUNDASHU = string
2
3 type HDCMS<T> = T extends XIANGJUNDASHU ? string : boolean
4
5 const hd: HDCMS<string> = '后盾人' //string
6
```

如果 extends 是泛型类型，并且传入的类型是联合类型。则分别进行判断，最后得到联合类型。

```
1 type XIANGJUNDASHU = string
2
3 type HDCMS<T> = T extends XIANGJUNDASHU ? string : boolean
4
5 const hd: HDCMS<string | number> = false //string | boolean
6
```

条件判断也可以嵌套使用

```
1 type XIANGJUNDASHU = string
2
3 type HOUDUNREN = string | number
4
5 type HDCMS<T> =
6  T extends XIANGJUNDASHU ? string :
7  T extends HOUDUNREN ? symbol : boolean
8
9 const hd: HDCMS<string | number> = '后盾人'
10
```

使用**[]**包裹类型，表示使用泛型的整体进行比较

```

```

```ts
type XIANGJUNDASHU = string | number

type HOUDUNREN = string | number

type HDCMS<T> = [T] extends [XIANGJUNDASHU] ? string : boolean

const hd: HDCMS<string | number> = '后盾人' //string

```

#Exclude

我们利用上面的泛型类型的条件分配，可以创建一个类型用于进行类型的过滤。

- 从 T 泛型类型 中过滤掉 U 的类型
- never 是任何类型的子类型，可以赋值给任何类型，没有类型是 never 的子类型或可以赋值给 never 类型(never 本身除外)

```ts
type EXCLUDE<T, U> = T extends U ? never : T

type XIANGJUNDASHU = string

type HOUDUNREN = string | number

const hd: EXCLUDE<HOUDUNREN, XIANGJUNDASHU> = 100; //number

```

事实上 typescript 已经提供了 Exclude 关键字用于完成上面的工作，所以我们不需要单独定义 Exclude 类型了。

```ts
type XIANGJUNDASHU = string

type HOUDUNREN = string | number

const hd: Exclude<HOUDUNREN, XIANGJUNDASHU> = 100;

```

#Extract

Extract 与 Exclude 相反，用于获取相交的类型。

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

type HOUDUNREN = string | number | boolean

const hd: EXTRACT<HOUDUNREN, string | number> = '后盾人';

```

下面是取两个类型相同的属性名

```ts
type HOUDUNREN = string | number | boolean

const hd: Extract<HOUDUNREN, string | number> = '后盾人';

```

#Pick

pick 可以用于从属性中挑选出一组属性，组成新的类型。

下面定义 pick 类型用于从 HOUDUNREN 类型中挑选出 name 与 age 类型。

```ts
type HOUDUNREN = { name: string, age: number, skill: string }
type PICK<T, U extends keyof T> = {
  [P in U]: T[P]
}

type HD = PICK<HOUDUNREN, 'name' | 'age'>
const xj: HD = { name: '后盾人', age: 33 }

```

同样 typescript 已经原生提供了 Pick 类型，所以我们不用像上面那样自己定义了

```ts
type HOUDUNREN = { name: string, age: number, skill: string }

type HD = Pick<HOUDUNREN, 'name' | 'age'>
const xj: HD = { name: '后盾人', age: 33 }

```

#Omit

从类型中过滤掉指定属性，这与 Pick 类型工具功能相反

```
type HD = { name: string, age: number, city: string }

type MyOmit<T, U> = Pick<T, {
  [K in keyof T]: K extends U ? never : K
}[keyof T]>

type XJ = MyOmit<HD, 'name' | 'age'> //{city:string}

```

可以将上面代码使用 Exclude 优化

```
type HD = { name: string, age: number, city: string }

type MyOmit<T, U> = Pick<T, Exclude<keyof T, U>>

type XJ = MyOmit<HD, 'name' | 'age'> //{city:string}

```

typescript 已经提供了类型工具 Omit

```
type HD = { name: string, age: number, city: string }

type XJ = Omit<HD, 'name' | 'age'> //{city:string}

```

#Partial

下面定义 Partial 类型，用于将全部属性设置为可选

```
type XIANGJUNDASHU = { name: string, age: number }

type PARTIAL<T> = {
 [P in keyof T]?: T[P]
}

```

```
const hd: PARTIAL<XIANGJUNDASHU> = { name: '向军' } // {name?:string,age?:number}

```

Typescript 原生提供了 Partial 的支持，所以我们不用自己定义了

```
type XIANGJUNDASHU = { name: string, age: number }

const hd: Partial<XIANGJUNDASHU> = { name: '向军' }

```

#Record

Record 常用于快速定义对象类型使用

下面我们来手动实现一个 Record，RECORD 类型的第一个参数为索引，第二个为类型

```
type RECORD<K extends string | number | symbol, V> = {
  [P in K]: V
}

type HD = RECORD<'name' | 'age', string | number>

const xj: HD = { name: "后盾人", age: 18 }

```

typescript 原生已经提供了 Record 类型，下面定义 MEMBER 类型，索引为字符串，值为任何类型

```
type HD = Record<'name' | 'age', any>

const xj: HD = { name: "后盾人", age: 18 }

```

#infer

- infer 只能在 extends 中使用
- infer 的类型变量，只能在 extends 条件的 true 中使用

下面使用 infer 推断属性值类型

```ts
type HD = { name: string, age: number }

type AttrType<T> = T extends { name: infer M, age: infer M } ? M : T

type valueType = AttrType<HD> //string | number

```

下面使用 infer 获取值类型

```ts
type USER = { name: string, age: number, get(a: string): void }

type GetType<T> = {
  [K in keyof T]: T[K] extends (infer U) ? U : K
}[keyof T]

type valueType = GetType<USER>;

```

下面是获取函数返回值类型

```ts
type HD = (n: string) => number[]

type GetFunctionReturnValue<T> = T extends ((...args: any) => (infer U)[]) ? U : T


type valueType = GetFunctionReturnValue<HD>;

```