24 条件类型，它不是三元操作符的写法吗？

更新时间：2019-07-04 11:06:51

> 成功 = 艰苦的劳动 + 正确的方法 + 少谈空话。
>
> ——爱因斯坦

3.11.1 基础使用

条件类型是 TS2.8 引入的，从语法上看它像是三元操作符。它会以一个条件表达式进行类型关系检测，然后在后面两种类型中选择一个，先来看它怎么写：

```
T extends U ? X : Y
```

这个表达式的意思是，如果 T 可以赋值给 U 类型，则是 X 类型，否则是 Y 类型。来看个实际例子：

```
type Type<T> = T extends string | number
let index: Type<'a'> // index的类型为string
let index2: Type<false> // index2的类型为number
```

3.11.2 分布式条件类型

当待检测的类型是联合类型，则该条件类型被称为“分布式条件类型”，在实例化时会自动分发成联合类型，来看例子：

```
type TypeName<T> = T extends any ? T : never;
type Type1 = TypeName<string | number>; // Type1的类型是string|number
```

你可能会说，既然想指定 Type1 的类型为 string|number，为什么不直接指定，而要使用条件类型？其实这只是简单的示范，条件类型可以增加灵活性，再来看个复杂点的例子，这是官方文档的例子：

```
type TypeName<T> = T extends string
 ? string
 : T extends number
 ? number
 : T extends boolean
 ? boolean
 : T extends undefined
 ? undefined
 : T extends Function
 ? Function
 : object;
type Type1 = TypeName<() => void>; // Type1的类型是Function
type Type2 = TypeName<string[]>; // Type2的类型是object
type Type3 = TypeName<(() => void) | string[]>; // Type3的类型是object | Function
```

我们来看一个分布式条件类型的实际应用：

```
type Diff<T, U> = T extends U ? never : T;
type Test = Diff<string | number | boolean, undefined | number>;
// Test的类型为string | boolean
```

这个例子定义的条件类型的作用就是，找出从 T 中出去 U 中存在的类型，得到剩下的类型。不过这个条件类型已经内置在 TS 中了，只不过它不叫 Diff，叫 Exclude，我们待会儿会讲到。

来看一个条件类型和映射类型结合的例子：

```
type Type<T> = { [K in keyof T]: T[K] extends Function ? K : never }[keyof T];
interface Part {
 id: number;
 name: string;
 subparts: Part[];
 updatePart(newName: string): void;
}
type Test = Type<Part>; // Test的类型为"updatePart"
```

来看一下，这个例子中，接口 Part 有四个字段，其中 updatePart 的值是函数，也就是 Function 类型。Type的定义中，涉及到映射类型、条件类型、索引访问类型和索引类型。首先[K in keyof T]用于遍历 T 的所有属性名，值使用了条件类型，T[K]是当前属性名的属性值，`T[K] extends Function ? K : never`表示如果属性值为 Function 类型，则值为属性名字面量类型，否则为 never 类型。接下来使用`keyof T`获取 T 的属性名，最后通过索引访问类型`[keyof T]`获取不为 never 的类型。

3.11.3 条件类型的类型推断-infer

条件类型提供一个`infer`关键字用来推断类型，我们先来看个例子。我们想定义一个条件类型，如果传入的类型是一个数组，则返回它元素的类型；如果是一个普通类型，则直接返回这个类型。来看下不使用 infer 的话，怎么写：

```
type Type<T> = T extends any[] ? T[number] : T;
type test = Type<string[]>; // test的类型为string
type test2 = Type<string>; // test2的类型为string
```

这个例子中，如果传入 Type 的是一个数组类型，那么返回的类型为`T[number]`，也就是该数组的元素类型，如果不是数组，则直接返回这个类型。这里我们是自己通过索引访问类型`T[number]`来获取类型的，如果使用 infer 关键字则无需自己手动获取，我们来看下怎么使用 infer：

```
type Type<T> = T extends Array<infer U> ? U : T;
type test = Type<string[]>; // test的类型为string
type test2 = Type<string>; // test2的类型为string
```

这里 infer 能够推断出 U 的类型，并且供后面使用，你可以理解为这里定义了一个变量 U 来接收数组元素的类型。

3.11.4 TS 预定义条件类型

TS 在 2.8 版本增加了一些预定义的有条件类型，来看一下：

- Exclude<T, U>，从 T 中去掉可以赋值给 U 的类型：

```
type Type = Exclude<"a" | "b" | "c", "a" | "b">;
// Type => 'c'
type Type2 = Exclude<string | number | boolean, string | number>;
// Type2 => boolean
```

- Extract<T, U>，选取 T 中可以赋值给 U 的类型：

```
type Type = Extract<"a" | "b" | "c", "a" | "c" | "f">;
// Type => 'a' | 'c'
type Type2 = Extract<number | string | boolean, string | boolean>;
// Type2 => string | boolean
```

- NonNullable，从 T 中去掉 null 和 undefined：

```
type Type = Extract<string | number | undefined | null>;
// Type => string | number
```

- ReturnType，获取函数类型返回值类型：

```
type Type = ReturnType<() => string)>
// Type => string
type Type2 = ReturnType<(arg: number) => void)>
// Type2 => void
```

- InstanceType，获取构造函数类型的实例类型：

InstanceType直接看例子可能不好理解，所以我们先来看下它的实现：

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

InstanceType 条件类型要求泛型变量 T 类型是创建实例为 any 类型的构造函数，而它本身则通过判断 T 是否是构造函数类型来确定返回的类型。如果是构造函数，使用 infer 可以自动推断出 R 的类型，即实例类型；否则返回的是 any 类型。

看过 InstanceType 的实现后，我们来看怎么使用：

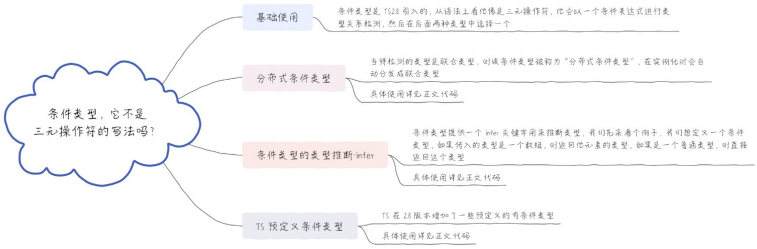
```
class A {
  constructor() {}
}
type T1 = InstanceType<typeof A>; // T1的类型为A
type T2 = InstanceType<any>; // T2的类型为any
type T3 = InstanceType<never>; // T3的类型为never
type T4 = InstanceType<string>; // error
```

上面例子中，T1 的定义中，`typeof A` 返回的的是类 A 的类型，也就是 A，这里不能使用 A 因为它是值不是类型，类型 A 是构造函数，所以 T1 是 A 构造函数的实例类型，也就是 A；T2 传入的类型为 any，因为 any 是任何类型的子类型，所以它满足 `T extends new (...args: any[]) => infer R`，这里 infer 推断的 R 为 any；传入 never 和 any 同理。传入 string 时因为 string 不能不给构造函数类型，所以报错。

本节小结

本小节我们学习了条件类型的相关知识，它的语法是`T extends U ? X : Y`，我们可以形象地理解它是三元操作符的形式，`T extends U`是判断条件，如果T的类型符合U，则取类型X，否则为类型Y。我们还学习了分布式条件类型，它比较简单，是条件类型的一种特殊情况，即待检测的类型是联合类型。我们还学习了如何使用**infer**来更好地利用类型推断。最后我们学习了几个TypeScript中常用的内置条件类型，方便我们开发使用。

下个小节我们将学习装饰器的基础部分，装饰器是实验性功能，ECMAScript对于装饰器的提案也是一再修改，截止到本专栏撰写时还没有定论，但是TypeScript已经实验性支持，你可以先体验下它。



23 前面跳过的unkown类型详解

25 入手装饰器，给凡人添加超能力