类型系统功能

乔恩·麦基

2021 年 9 月 3 日 警察

强大的类型系统功能介绍。

Typescript 类型系统比表面上看起来要强大得多。本文介绍了它的一些高级功能以及一些基础知识，以展示您使用类型系统的灵活性和创造性。

两个示例使用本文中的许多功能来增加类型系统中的数字：

下载示例.zip

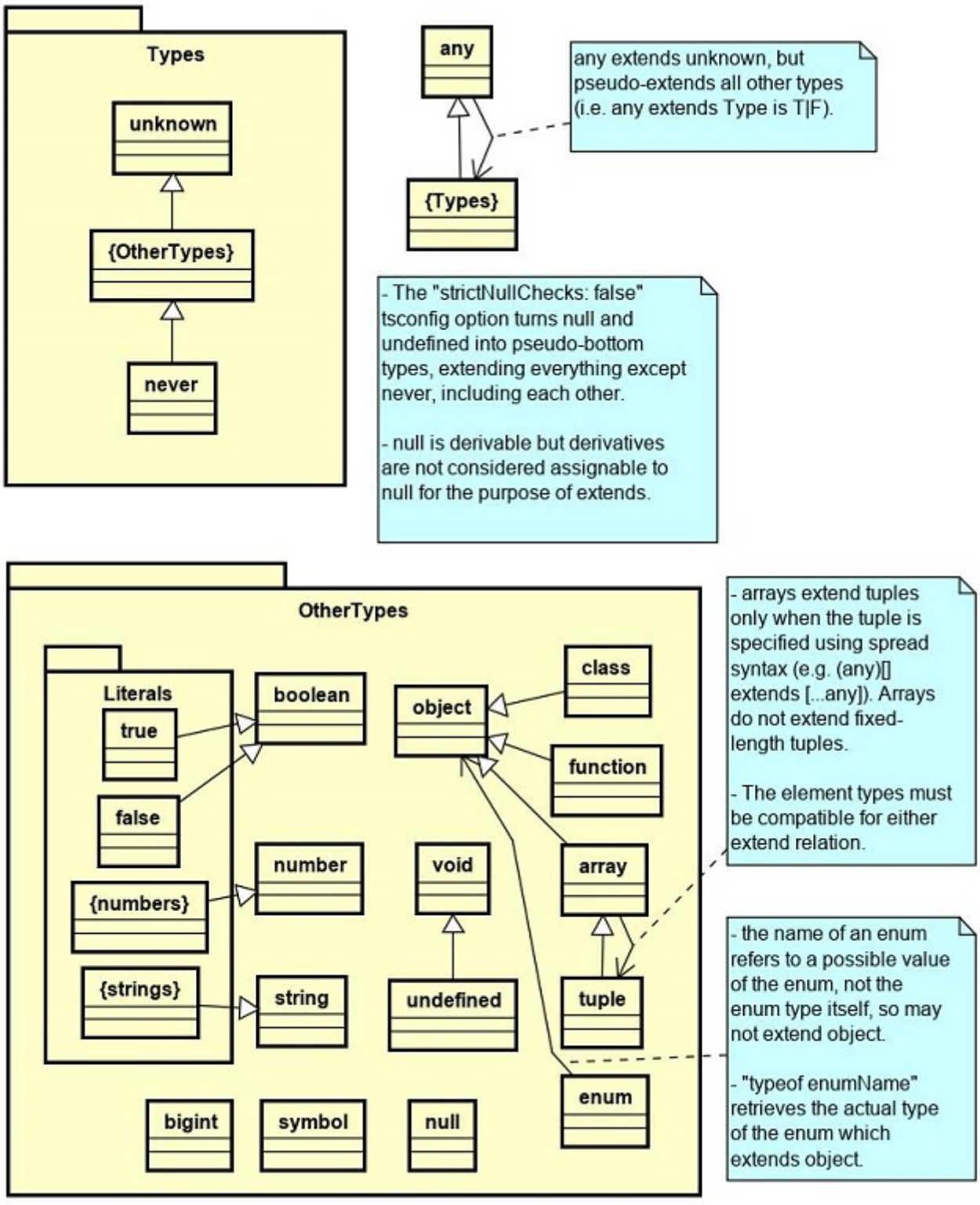
内容

介绍

我听说开发人员将 Typescript 称为“带有类型的 Javascript”，一开始我也犯了同样的错误。虽然从广义上讲这可能有点正确，但您获得的类型系统比表面上看起来功能丰富得多。它提供了基本的逻辑构造、对象转换和许多可能是无意的技巧，让您比大多数人可能习惯于来自大多数流行语言的人更具表现力。

基本类型

乍一看，Typescript 类型系统相对于基本类型如下所示：



它可能看起来很复杂，但如果你盯着它看一会儿，它就很有意义。关系读作“`undefined`扩展`void`”或等效地“`undefined`可分配给” `void`。快速总结：

- `any`是通用型。它是顶部类型，底部类型，以及介于两者之间的所有类型。
- `unknown`是顶级类型。一切都可以分配给`unknown`。

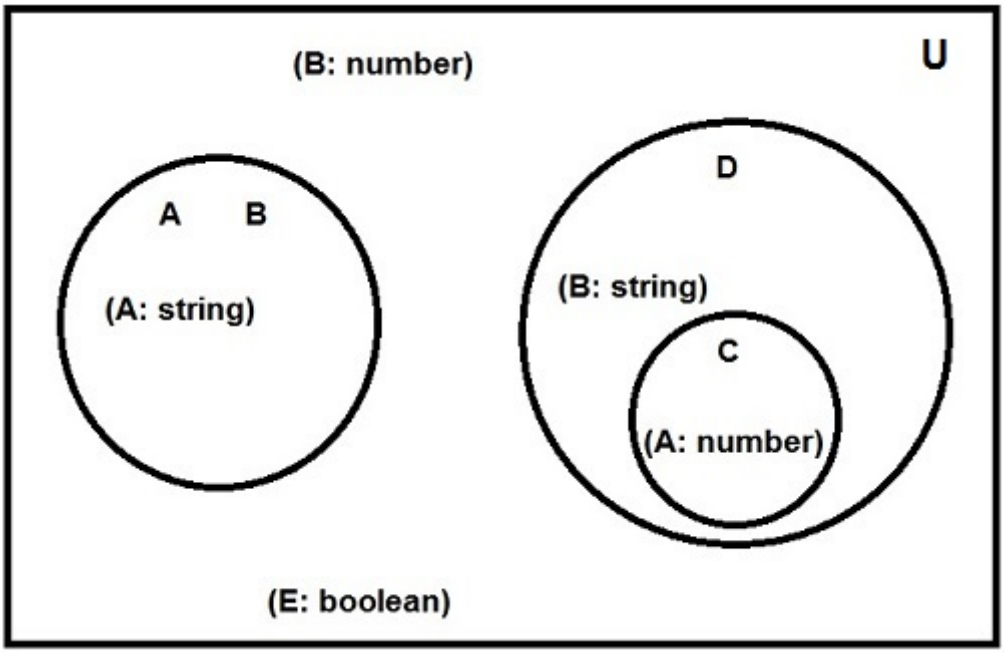
- never是底部类型。never可分配给一切。
- 文字扩展它们的非文字对应物（例如trueextends boolean、3extends number）。
- 复杂的基本类型扩展object（例如function, tuple, enum）

不同的仅行箭头适用于存在一些显着异常的情况，例如数组和元组之间的可分配性。请记住，虽然这些看起来像是继承关系，但事实并非如此，因为 Typescript 使用结构类型系统。

打字稿 复制代码

```
class A { A: string; } class B { A: string; }
var a: A extends B ? true : false; //true
```

该类型的a是true，尽管事实上A并B在层次无关。这是因为它们具有相同的结构。在我看来，从集合的角度思考要容易得多：



打字稿 复制代码

```
class A { A: string; }
class B { A: string; }
class C { A: number; }

class D { A: number; B: string; }
//or equivalently
class D extends C { B: string; }
```

从集合的角度来看，类型可以被认为是 U所有可能属性的通用集合中命名的属性分组，其中属性是与类型相关联的名称。因此A: number和A: string是不同的性质以及A: string和B: string。如果一个类型是另一种类型的超集，则它会扩展另一种类型。所以在上面的例子中，Dextends C。如果超集关系不是适当的超集，则两种类型都会相互扩展 - AextendsB和Bextends A。这是因为它们实际上是相同类型的不同名称，因为它们具有完全相同的属性集。

然而它最终是有道理的，重要的部分是它是有道理的，因为其余的特性需要理解类型之间是如何关联的。

特征

A 扩展 B ? 真分支：假分支

这相当于一个三元组if，条件是从Ato 的赋值兼容性B。此外，在真正的分支中，A的类型具有B类似于类型保护工作方式的附加类型约束。如果A是一个裸类型参数（只是一个单独的类型变量）并且是一个联合，则该联合分布在 上extends并且结果联合在一起。例如：

打字稿 复制代码

```
class A { A: string; }
class B extends A { B: string; }
class C extends B { C: string; }

type example<T> = T extends B ? true : false;
var a: example<A|C>; //a has the type boolean (i.e. true | false)
```

发生上述情况是因为在分布后我们得到`(A extends B ? true : false)|(C extends B ? true : false)`简化为`false | true`。请注意，在以下情况下不会发生这种情况：

打字稿 复制代码

```
var b: A|C extends B ? true : false; //b has the type false.
```

由于我们不再有类型参数，因此不会发生分布。还要考虑以下情况：

打字稿 复制代码

```
type example2<T> = [T] extends [B] ? true : false;
var c: example2<A|C>; //c has the type false.
var d: example2<C>; //d has the type true.
```

类型参数不再是裸露的，因为我们在元组类型（`[T]`)中使用它，因此不会发生分布。

推断的类型变量

推断类型变量是类似于类型参数的临时类型变量，但绑定到真正分支的范围内，并由上下文自动分配。它们的类型与可以安全推断的范围一样窄/特定（否则所有推断都将简单地解析为any或unknown）。

打字稿 复制代码

```
type example<T> = T extends infer U ? U : never;
var a: example<boolean>; //boolean
```

一个更复杂的例子：

打字稿 复制代码

```
type example<T extends [...any]> = T extends [...infer _, infer A] ? A : T;
var a: example<[boolean, null, undefined, number]>; //number
var b: example<[void]>; //void
var c: example<[]>; //unknown
```

这使用推断类型变量返回元组中的最后一个类型。请注意，我们正在本质上说，这里是“我们可以推断出成功的类型_，并A从T给定签名有意义吗？”推理语句的签名会对结果变量的类型产生重要影响：

打字稿 复制代码

```
type example<T extends [...any]> = T extends (infer A)[] ? A : never;
var a: example<[string, number]>; //string|number

type example2<T extends [...any]> = T extends [...infer A] ? A : never;
var b: example2<[string, number]>; //[string, number]
```

它们之间的主要区别在于，它`(infer A)[]`是一个可变长度的数组签名，`[...infer A]`一个是一个固定长度的数组签名（即元组）。这显示了结果类型信息如何在第一个示例中扩展为联合，因为它是从可变长度数组签名推断出来的，尽管输入是更窄的元组类型。之所以会丢失位置类型信息，是因为可变长度数组签名没有位置概念——只有哪些类型对其所有（不是每个）元素都有效。

立即索引对象定义（即类型的 switch 语句）

一个限制`extends`是它不能完全允许开关式分支。我们可以通过使用这样一个事实来解决这个问题：如果你索引一个对象定义，你得到的是关联属性的类型，它有效地将对象定义转换为 switch 语句：

打字稿 复制代码

```
type PickOne<Choice extends 0 | 1 | 2 | 3> =
    {
        0: 'Uhhh...',
        1: 'Odd',
        2: 'Even',
        3: 'Odd'
    } [Choice];
```

请记住，文字是一种类型！如果我们只是使用`extends`它看起来像：

打字稿 复制代码

```
type PickOne<Choice extends 0 | 1 | 2 | 3> =
    Choice extends 0 ?
        'Uhhh...'
        : Choice extends 1 ?
            'Odd'
            : Choice extends 2 ?
                'Even'
                : Choice extends 3 ?
                    'Odd'
                    : never;
```

有点乱吧？对象索引器还支持产生有效索引的任何类型表达式——不仅仅是简单的类型参数：

打字稿 复制代码

```
type Nullable<T> =
    {
        0: T | null,
        1: T
    } [T extends string | object ? 0 : 1];
```

一个有点愚蠢的例子，但它展示了这一点。另一个巧妙的功能是支持递归：

打字稿 复制代码

```
type digit = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9;
type DigitsAfter<X extends digit, Current = never> =
    {
        0: DigitsAfter<1, 1>,
        1: DigitsAfter<2, Current | 2>,
        2: DigitsAfter<3, Current | 3>,
        3: DigitsAfter<4, Current | 4>,
        4: DigitsAfter<5, Current | 5>,
        5: DigitsAfter<6, Current | 6>,
        6: DigitsAfter<7, Current | 7>,
        7: DigitsAfter<8, Current | 8>,
        8: Current | 9,
        9: never
    } [X];
```

只要递归不能无限递归，它就可以工作。如果无限递归是可能的，则在使用类型或递归属性时会出现错误，从而导致问题。有时这对类型系统接受的内容可能有点挑剔，但如果您有明确的基本情况和浅递归，它往往效果最佳。

方差

Typescript 本身的差异并没有真正不同。预期的标准东西在那里：

复制代码

```
class A { A: string; }
class B extends A { B: string; }
class C extends B { C: string; }

let f1: ()=>A = () => new B(); //Covariant with respect to return types
let f2: (_: B)=>void = (_: A) => {}; //Contravariant with respect to parameter types
```

```
let f3: (_: B)=>B = (_: A) => new C(); //Mixed variance

let f4: (_:B)=>void = (_:C) => {}; //Wait... what? Bivariant parameter type?
```

如果"strictFunctionTypes":false设置了tsconfig 选项，则对标准内容有一个重要补充- 函数参数类型是双变量的。在我的经验中，除了在双变量函数上下文中明确过滤事物之外，并不是特别有用，但在需要时要注意这一点。

打字稿 复制代码

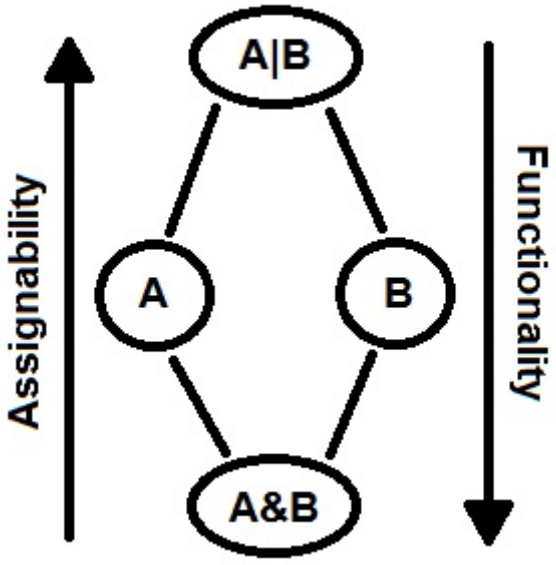
```
//Note: requires '"strictFunctionTypes": false' to be set in tsconfig
class A { A: string; }
class B extends A { B: string; }
class C extends B { C: string; }
class D { D: string; }
class E extends D { E: string; }

type GetTypesRelatedTo<T, options> =
    T extends any ? //distribute T if it's a union to handle each type separately
        options extends any ? //distribute options
            //compare each option to T in a bivariant context
            ((_:options)=>void) extends ((_:T)=>void) ?
                options
                : never
            : never
        : never;

let test: GetTypesRelatedTo<B, A | B | C | D | E>; //returns B | A | C
```

函数联合和交集

联合和交集的行为分别与基类和派生类非常相似。与继承非常相似，可分配性和功能性的属性彼此相反。"向上"朝向更多的基本类型会产生更大的可分配性但功能更低，而"向下"朝向更多派生类型会产生更低的可分配性但功能更强大。



从这个角度来看，联合是相对于其组成类型的基本类型，而交集是相对于其组成类型的派生类型。我们可以验证这个类比成立：

打字稿 复制代码

```
class A { A: string; }
class B { B: string; }
class AB { A: string; B: string; }

let x: A | B;
//Testing assignability
x = new A();
x = new B();
x = new AB();
//Testing functionality
```

```
x.A;          //Error, property A does not exist on type A | B
x.B;          //Error, property B does not exist on type A | B

let y: A & B;
//Testing assignability
y = new A();  //Error, type A is not assignable to type A & B
y = new B();  //Error, type B is not assignable to type A & B
y = new AB();
//Testing functionality
y.A;
y.B;
```

这个类比对于讨论函数的并集和交集特别方便。例如，尝试填写"???" 将以下内容强制转换为T=>U形式时：

打字稿 复制代码

```
(a: ???)=>??? = ((a: string)=>string) | ((a: number)=>number)

(a: ???)=>??? = ((a: string)=>string) & ((a: number)=>number)
```

为了解决这个问题，首先让我们确定回报是多少，因为这是一个简单的起点：

1. 函数的联合自然会有一个返回类型，它是函数返回类型的联合。
2. 函数的交集自然会有一个返回类型，即函数返回类型的交集。

在考虑赋值时，这导致了一个有点令人惊讶的结论；强制转换为某种T=>U形式并不会产生严格等效的结果。相反，函数联合的强制创建一个基本类型，而函数交集的强制创建一个更派生的类型。

打字稿 复制代码

```
//If this was reversed, (string | number) wouldn't be assignable to string due to possibility
//of getting a number, and it wouldn't be assignable to number due to the possibility of
getting
//a string.
(a: ???)=>(string | number) = ((a: string)=>string) | ((a: number)=>number)

//If this was reversed, string wouldn't be assignable to (string & number) due to not having
//the properties of number, and number wouldn't be assignable to it either due to not having
//the properties of string.
((a: string)=>string) & ((a: number)=>number) = (a: ???)=>(string & number)
```

旁注：尽管只讨论类型之间的可赋值性，但重要的是要记住类型的目的是描述具体的表示。在确定什么应该兼容时，必须考虑这一事实。虽然联合示例相对直观，但即使在交集示例中我们也考虑单独分配可能会令人惊讶。这是因为实际上，函数交集代表了具体级别的函数重载，因此具体结果仅来自单个函数。这就是为什么string=>string & number=>number不能分配给???=>(string & number). 实际上只会返回这些值中的一个，并且它的返回不会满足更多的导数(string & number)。以下是这些属性的示例：

打字稿 复制代码

```
class Overload {
    static test(a: string): string;
    static test(a: number): number;
    static test(a: string | number) { return a; }
}
//Behavior of the function overload
let overloadTest1 = Overload.test("a"); //variable type is string
let overloadTest2 = Overload.test(5); //variable type is number

//Behavior of the function intersection (concretely backed by the function overload)
let overloadFunc: ((a:string)=>string) & ((a:number)=>number) = Overload.test;
let overloadTest3 = overloadFunc("a"); //same result as overloadTest1
let overloadTest4 = overloadFunc(5); //same result as overloadTest2
```

现在我们需要找出原始示例的参数类型。对于联合，我们想要更多的导数（参数的逆变），因此我们将采用，string & number因为它可以分配给stringor number。对于路口，我们要少衍生因为我们正在处理的任务的反面现在这样我们就一起去的东西string | number，因为这两个string和number可分配给它。

打字稿 复制代码

```
(a: string & number)=>(string | number) = ((a: string)=>string) | ((a: number)=>number)

((a: string)=>string) & ((a: number)=>number) = (a: string | number)=>(string & number)
```

总之，将函数的交集强制转换为T=>U形式会创建更派生的类型，而对函数联合执行相同的操作会创建基本类型。下面展示了我们刚刚想到的一切：

打字稿 复制代码

```
class W { w: string; }
class U { u: string; }

declare const Inter: ((a: W)=>W) & ((a:U)=>U);
declare const TU_Inter: (a: W | U)=>(W & U);
let ex1: typeof Inter = TU_Inter;
let ex2: typeof TU_Inter = Inter; //Error

declare const Union: ((a: W)=>W) | ((a:U)=>U);
declare const TU_Union: (a: W & U)=>(W | U);
let ex3: typeof Union = TU_Union; //Error
let ex4: typeof TU_Union = Union;
```

在下面的示例中，我们将逐步解释如何使用函数联合和交集的属性从联合中提取最后一个元素（参见本节末尾的注释）的代码；在不知道元素类型以区分联合的情况下，从表面上看似乎是不可能的事情。

打字稿 复制代码

```
type UnionToFunctionIntersection<T> =
    (T extends any ? () => T : never) extends infer U ?
        //Function union coerced into the form V=>void.
        (U extends any ? (_: U) => void : never) extends (_: infer V) => void ?
            V
            : never
        : never;
//Function intersection coerced into the form ()=>U.
type Last<T> = UnionToFunctionIntersection<T> extends () => (infer U) ? U : never;
```

让我们从顶部分解它。

打字稿 复制代码

```
type UnionToFunctionIntersection<T> =
    (T extends any ? () => T : never) extends infer U ?
```

我们在这里做的是分发T，以防它是联合，然后返回一个函数联合，其中每个元素都T处于返回类型位置。我们正在推断这个结果，U所以我们有一个简单的类型变量供以后引用。这也意味着下一行代码将正确分配我们刚刚创建的联合。

打字稿 复制代码

```
(U extends any ? (_: U) => void : never) extends (_: infer V) => void ?
```

在这里，我们分发函数的联合U，然后返回另一个函数联合，每个元素都U在参数类型位置。例如，如果我们开始与类型A | D的T，U将是()=>A | ()=>D，现在的签名会(_:()=>A)=>void | (_:()=>D)=>void。看起来很可笑，但这都是下一部分的设置。

因此，当我们到达 时extends (_: infer V) => void，乍一看似乎我们只是找回了原始值，U因为我们在将 的元素移入的同一位置推断了一个类型变量U。但是，仅当表达式是简单类型变量时才会发生类型分布。之前的复杂表达式extends不是简单的类型变量。因此，如果不发生分布，那么可以V推断出复杂表达式是(_:()=>A)=>void | (_:()=>D)=>void什么？

打字稿 复制代码

```
//A reminder of the current state of the expression we're exploring
((_:()=>A)=>void | (_:()=>D)=>void) extends (_: infer V) => void
```

如果您还记得本节前面的联合讨论，将函数的联合强制转换为T=>Uform 会导致参数类型相交，因此V推断为()=>A & ()=>D。

现在让我们看看最后一行，看看为什么我们经历了所有这些麻烦只是为了将联合变成函数的交集。

打字稿 复制代码

```
type Last<T> = UnionToFunctionIntersection<T> extends () => (infer U) ? U : never;
```

在这里，我们采用函数交集并推断其返回类型的类型变量。的返回类型是()=>A & ()=>D什么？同样，如果您还记得之前的讨论，将函数的交集强制转换为T=>Uform 会导致返回类型相交，因此U推断为A & D。除了记住可分配性是从左到右检查的extends。我们往反方向走！我们解决了T=>U更早派生形式的一般情况，但这里需要较少派生。

打字稿 复制代码

```
()=>U = (()=>A) & (()=>D)
```

由于没有参数，我们只需要从标准协变的角度考虑答案。这跑进一个小问题，虽然因为一个路口是分配给它的contituent类型-无论是A和D工作U。这里没有逻辑上最好的解决方案。只需要选择一个，因此 Typescript 决定始终返回交集中的最后一个返回类型。U因此推断为D。这也有副作用，即在 Typescript 中交集不是严格可交换的（尽管它们在许多上下文中看起来像）。

打字稿 复制代码

```
declare const F1: ((a:string)=>string) & ((a:number)=>number);
declare const F2: ((a:number)=>number) & ((a:string)=>string);

//Non-commutative
type F1Last = typeof F1 extends (_:any)=>(infer L) ? L : never; //number
type F2Last = typeof F2 extends (_:any)=>(infer L) ? L : never; //string

//Another example of how intersections represent overloading.
F1('a'); //in my editor, this shows as "(a:string)=>string (+1 overload)".
F1(1); //this shows as "(a:number)=>number (+1 overload)".
//Same results as above if you use F2
```

这结束了所有的黑魔法。我们构建了一个看起来很可笑的函数签名，以便利用独特的属性来扭转和解开它，即如何为函数联合推断参数类型以及如何为函数交叉推断返回类型，以便从联盟。

重要说明：实践中的联合通常是有序的，但 Typescript 规范不要求这是真的。因此，只能安全地假设Last<T>将从联合中提供**某种**类型而不是**最后**一种类型，即使在实践中后者在大多数情况下最终都是正确的。这是因为用于检索最后一个类型的交集基于联合，因此联合排序的更改将更改结果交集的排序。

分解

分解是将元组分解为各个部分的过程。实现这一点的最简单方法是使用带有元组签名的推理，该元组签名对未显式处理的元素使用 rest 参数。

打字稿 复制代码

```
type FirstElementOf<Tuple> = Tuple extends [infer E, ...infer _] ? E : never;
type LastElementOf<Tuple> = Tuple extends [...infer _, infer E] ? E : never;
```

您也可以为此使用函数上下文：

打字稿 复制代码

```
type FirstElementOf<Tuple> =
    (Tuple extends any[] ? (...a:Tuple)=>void : never) extends (a:infer E, ...b:infer _)=>void
?
        E
        : never;
```

函数在参数列表的开头不支持剩余参数，因此LastElementOf<Tuple>不能以这种方式实现。这种方法与更清洁、更简单的第一种方法相比没有优势；这只是一个有趣的脚注。

映射类型

正如我们在前几节中看到的，该`extends`子句对于使用现有类型创建新类型非常有用。例如，`A extends any ? ()=>A : never`或`A extends any ? { a: A } : never`。Typescript 还具有使用现有类型的属性而不是整个类型的映射类型来创建新类型的机制。

打字稿 复制代码

```
//Basic structure
{
    [ <propertyName> in <properties> as <propertyRemap> ]: <propertyType>
}
```

`propertyName`是一个变量，用于保存从 set 返回的每个元素`properties`。它的工作方式与 Javascript`for...of`循环的工作方式相同。`propertyRemap`是一个可选表达式，您可以在其中转换`propertyName`为其他一些有效值以用作属性索引。`propertyType`是新类型的属性。让我们看一些基本的例子，看看为什么所有这些都很有用。

下面的示例在`Lowercase`映射类型中使用内置实用程序类型来创建一个新类型，该类型是所有属性名称均小写的旧类型。

打字稿 复制代码

```
type LowercaseProperties<T> =
    {
        [Key in keyof T as Lowercase<Key>]: T[Key]
    };

type TestType = { ABC: number; XYZ: string };
type TestTypeLower =  LowercaseProperties<TestType>; //{ abc: number; xyz: string }
```

不过，这只是它能力的一小部分。您可以从原始信息创建新类型：

打字稿 复制代码

```
type Create<PropertyNames extends string> =
    {
        [Key in PropertyNames as
            (Key extends 'id' ? Uppercase<Key> : Capitalize<Lowercase<Key>>)
        ]: Key extends 'id' ? number : string
    };

type IdCardType = Create<'id'|'firstName'|'lastName'>;
/*
{
    ID: number;
    Firstname: string;
    Lastname: string;
}
*/
```

您还可以通过在修饰符前使用 + 或 - 来使用它来更改类型的只读和可选修饰符。下面创建一个具有所有必需属性的只读类型。

打字稿 复制代码

```
type MakeReadonly<T> =
    {
        +readonly [Key in keyof T]-?: T[Key]
    };

type ReadonlyID = MakeReadonly<IdCardType>;
```

这也表明，如果`as`省略可选子句，则迭代器变量将用于属性索引。

模板化字符串文字

正是它在罐头上所说的。小心，因为它们在您需要时很方便，但插值位置是交叉相乘的，因此可能会迅速失控，导致编译器崩溃，可怕的联合。例如，以下模板创建从 0 到 1999 的每个数字。添加更多数字并观看 Typescript，希望它永远不会被创建。

打字稿 复制代码

```
type Digit = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9;
type UhOh = `${0|1}${Digit}${Digit}${Digit}`;
```

在映射类型时，模板通常可用于从旧的属性索引创建新的属性索引。

打字稿 缩小▲ 复制代码

```
//Look ma! I'm a real Java bean now!
type MakeJava<T> =
    {
        -readonly [K in keyof T]-?: T[K]
    } &
    {
        -readonly [K in keyof T as `get${Capitalize<Lowercase<K>>}`]: ()=>T[K]
    } &
    {
                    //Skip mutator for any property named 'ID'
        -readonly [K in Exclude<keyof T, 'ID'> as `set${Capitalize<Lowercase<K>>}`]
            : (set:T[K])=>void
    };

//Uses the ReadonlyID type from the previous section.
type JavaID = MakeJava<ReadonlyID>;
/* Resulting object
{
    ID: number;
    Firstname: string;
    Lastname: string;

    getID: ()=>number;
    getFirstname: ()=>string;
    getLastname: ()=>string;

    setFirstname: (set: string)=>void;
    setLastname: (set: string)=>void;
}
*/
```

这就是他们的全部。

延迟类型解析

我什至不会假装完全理解某些类型解析行为是如何或为什么发生的。但是，无论出于何种原因，似乎确实存在类型解析总是延迟的情况，这对于帮助处理由 Typescript 的递归深度限制引起的"类型实例化太深"错误非常有用。每当类型别名是对象定义的属性的类型并且该对象定义从别名返回时，在返回之前似乎不需要解析类型别名。如果我们认为我们不需要解析属性类型来确定要返回的对象类型，那么这是有道理的。

在行动中展示这一点可能是最简单的：

打字稿 缩小▲ 复制代码

```
type Digit = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9;
type FiftyNumbers = `${0|1|2|3|4}${Digit}`;

type UnionToFunctionIntersection<T> =
    (T extends any ? () => T : never) extends infer U ?
        (U extends any ? (_: U) => void : never) extends (_: infer V) => void ?
            V
            : never
        : never;

type Last<T> = UnionToFunctionIntersection<T> extends () => (infer U) ? U : never;

type Push<Tuple extends unknown[], Element> = [...Tuple, Element];

type GenerateTuple<
    Union,
    IsEmpty extends boolean = [Union] extends [never] ? true : false
```

```
> =
    true extends IsEmpty ?
        []
        : Last<Union> extends infer T ?
            GenerateTuple<Exclude<Union, T>> extends [...infer U] ?
                Push<U, T>
                : never
            : never;

//Error - Type instantiation is excessively deep and possibly infinite. ts(2589)
type tuple = GenerateTuple<FiftyNumbers>;
```

以上代码所做的就是使用给定联合的元素创建一个元组。我们知道类型实例化不是无限的，因为只有 50 个元素，所以为了避免错误，我们需要做的就是避免递归深度限制。为此，我们将重写GenerateTuple以使用延迟类型解析。

打字稿 复制代码

```
type PushFront<Tuple extends unknown[], Element> = [Element, ...Tuple];

type _GenerateTuple<
    Union,
    Tuple extends unknown[] = [],
    IsEmpty extends boolean = [Union] extends [never] ? true : false
> =
    true extends IsEmpty ?
        { _wrap: Tuple }
        : Last<Union> extends infer T ?
            { _wrap: _GenerateTuple<Exclude<Union, T>, PushFront<Tuple, T>> }
            : never;
```

原始GenerateTuple元素从末尾剥离联合元素，然后在递归展开时从这些元素中创建元组。由于 new_GenerateTuple使用递归来创建深度嵌套的对象定义，我们将使用稍微不同的方法来创建元组 - 在结束时而不是结束时。因此，我们需要将元素推到元组的前面以保持顺序，并且我们需要一个新的类型参数来将元组传递给下一次调用，因为我们不能再依赖堆栈返回。生成的对象将如下所示：

打字稿 复制代码

```
//abridged for my own sanity
{ _wrap: { _wrap: { _wrap: { _wrap: { _wrap: ['00', '01', '02', '03', '04'] } } } } }
```

所以我们已经解决了递归深度限制问题，但似乎我们已经把它换成了另一个问题——我们如何以与深度无关的方式解开这个问题？幸运的是，这并不像乍一看那样有问题。

打字稿 复制代码

```
type Unwrap<T> = T extends { _wrap: unknown } ? Unwrap<_Unwrap<T>> : T;
type _Unwrap<T> =
    T extends { _wrap: { _wrap: infer R } } ?
        { _wrap: _Unwrap<R> }
        : T extends { _wrap: infer R } ?
            R
            : T;

type GenerateTuple<Union> = Unwrap<_GenerateTuple<Union>>;
```

让我们首先解决核心别名 - _Unwrap。前两行所做的是将两个_wrap对象折叠成一个_wrap对象，_Unwrap递归调用剩下的任何对象。我们在这里再次使用延迟类型解析，以确保我们在展开时不会达到深度限制。一旦没有两个_wrap要折叠的对象，就必须只有一个，所以我们只需打开它。如果在_Unwrap非包装值上被调用，我们只返回该值（T）。

因此，_Unwrap当完全解决时的单个“外部”调用，包括它自己的递归调用，会导致_wrap整个结构中的对象数量减半。为了完全解开这个值，我们需要调用_Unwrap直到我们完全解开内部值。这就是Unwrap它的作用。它调用_Unwrap然后设置一个递归调用到它自己的结果。只有当该结果不再与_wrap对象的结构匹配时（即结果被完全展开），它才会返回一个将作为内部值的结果。

打字稿 复制代码

```
type example = { _wrap: { _wrap: { _wrap: { _wrap: { _wrap: ['00', '01', '02', '03',
'04']}}}}};

type unwrappedType = Unwrap<example>; //resulting type is ['00', '01', '02', '03', '04']
```

中提琴！使用延迟类型解析，我们成功避免了深度限制错误并执行了操作。

同样，这是对我所谓的延迟类型解析在幕后到底发生了什么的有根据的猜测，但我的猜测是有效发生的是，由于属性类型不需要具有完整的对象类型，因此对象类型是标记完成返回，别名展开，然后在返回对象后，系统立即意识到属性类型未完全解析，然后触发对象内的别名被调用。这意味着通过使用这种技术，我们实际上只会使用超出我们当前处理递归别名的单个堆栈深度。这也意味着不幸的是，我们无法使包装“更干净”，因为用于包装的别名会在返回时触发内部属性类型的解析，

打字稿 复制代码

```
type Digit = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9;

type FiftyNumbers = `${0|1|2|3|4}${Digit}`;

//The "clean" way to wrap objects, abstracting out the property name details.
type Wrap<T> = { _wrap: T };

type GenerateTuple2<Union> = Unwrap<_GenerateTuple2<Union>>;

//All we need to do to break the solution is replace the raw object wrapping
//that is recursive with the Wrap<> call.
type _GenerateTuple2<
    Union,
    Tuple extends unknown[] = [],
    IsEmpty extends boolean = [Union] extends [never] ? true : false
> =
    true extends IsEmpty ?
        { _wrap: Tuple } //Replacing this with Wrap<Tuple> would work because it only
                         //executes once since it's the base case for our recursion.
        : Last<Union> extends infer T ?
            //This blows up though.
            Wrap<_GenerateTuple2<Exclude<Union, T>, PushFront<Tuple, T>>>
            : never;

//Uh oh...
type tuple2 = GenerateTuple2<FiftyNumbers>;
```

谢谢阅读！

参考

打字稿手册

GitHub：Typescript 递归条件类型 PR#40002

GitHub: Typescript 函数交叉问题#42204

Susisu：如何创建深度递归类型

历史

21 年 9 月 2 日：初始版本。

执照

分享

关于作者

乔恩·麦基
软件开发人员
美国

手表
该会员

软件工程师致力于不断学习和改进，专注于自我记录代码和明智的设计决策。对大多数东西都感兴趣。

评论和讨论

添加评论或问题 

电子邮件提醒 Search Comments

-- 本论坛暂无消息 --

永久链接
广告
隐私
Cookie
使用条款

布局：固定 | 体液