Tipos Condicionais

No coração da maioria dos programas úteis, temos que tomar decisões com base na entrada. Programas em JavaScript não são diferentes, mas dado o fato de que os valores podem ser facilmente inspecionados, essas decisões também se baseiam nos tipos das entradas. Tipos condicionais ajudam a descrever a relação entre os tipos de entradas e saídas.

```
interface Animal {
  live(): void;
}
interface Dog extends Animal {
  woof(): void;
}

type Exemplo1 = Dog extends Animal ? number : string;

type Exemplo1 = number;

type Exemplo2 = RegExp extends Animal ? number : string;

type Exemplo2 = string;
```

Tipos condicionais têm uma forma que se parece um pouco com expressões condicionais (`condição ? expressãoVerdadeira : expressãoFalsa`) em JavaScript:

```
  SomeType extends OtherType ? TrueType : FalseType;
```

Quando o tipo à esquerda do `extends` é atribuível ao da direita, você obterá o tipo no primeiro ramo (o ramo "verdadeiro"); caso contrário, você obterá o tipo no ramo seguinte (o ramo "falso").

A partir dos exemplos acima, os tipos condicionais podem não parecer imediatamente úteis - podemos nos perguntar se `Dog extends Animal` e escolher `number` ou `string`! Mas o poder dos tipos condicionais vem do uso com genéricos.

Por exemplo, vamos considerar a seguinte função `createLabel`:

```
interface IdLabel {
  id: number /* alguns campos */;
}
interface NameLabel {
  name: string /* outros campos */;
}

function createLabel(id: number): IdLabel;
function createLabel(name: string): NameLabel;
function createLabel(nameOrId: string | number): IdLabel | NameLabel;
function createLabel(nameOrId: string | number): IdLabel | NameLabel {
  throw "não implementado";
}
```

Essas sobrecargas para createLabel descrevem uma única função JavaScript que faz uma escolha com base nos tipos de suas entradas. Note algumas coisas:

1. Se uma biblioteca precisa fazer o mesmo tipo de escolha repetidamente em sua API, isso se torna cansativo.

2. Precisamos criar três sobrecargas: uma para cada caso em que temos certeza do tipo (uma para `string` e uma para `number`), e uma para o caso mais geral (aceitando um `string | number`). Para cada novo tipo que `createLabel` pode manipular, o número de sobrecargas cresce exponencialmente.
   Em vez disso, podemos codificar essa lógica em um tipo condicional:

```
type NameOrId<T extends number | string> = T extends number
  ? IdLabel
  : NameLabel;
```

Podemos então usar esse tipo condicional para simplificar nossas sobrecargas em uma única função sem sobrecargas.

```
function createLabel<T extends number | string>(idOrName: T): NameOrId<T> {
  throw "não implementado";
}

let a = createLabel("typescript");

let a: NameLabel;

let b = createLabel(2.8);

let b: IdLabel;

let c = createLabel(Math.random() ? "hello" : 42);
let c: NameLabel | IdLabel;
```

Restrições de Tipo Condicional

Frequentemente, as verificações em um tipo condicional nos fornecerão algumas novas informações. Assim como o estreitamento com guardas de tipo pode nos dar um tipo mais específico, o ramo verdadeiro de um tipo condicional restringirá ainda mais os genéricos pelo tipo que estamos verificando.

Por exemplo, vamos considerar o seguinte:

```
type MessageOf<T> = T["message"];
Tipo '"message"' não pode ser usado para indexar o tipo 'T'.
```

Neste exemplo, o TypeScript retorna um erro porque `T` não é conhecido por ter uma propriedade chamada `message`. Poderíamos restringir `T`, e o TypeScript não reclamaria mais:

```
type MessageOf<T extends { message: unknown }> = T["message"];

interface Email {
  message: string;
}

type ConteudoMensagemEmail = MessageOf<Email>;

type ConteudoMensagemEmail = string;
```

No entanto, e se quiséssemos que `MessageOf` aceitasse qualquer tipo e retornasse algo como `never` se uma propriedade `message` não estivesse disponível? Podemos fazer isso movendo a restrição para fora e introduzindo um tipo condicional:

```
type MessageOf<T> = T extends { message: unknown } ? T["message"] : never;

interface Email {
  message: string;
}

interface Dog {
  bark(): void;
}

type ConteudoMensagemEmail = MessageOf<Email>;

type ConteudoMensagemEmail = string;

type ConteudoMensagemDog = MessageOf<Dog>;

type ConteudoMensagemDog = never;
```

Dentro do ramo verdadeiro, o TypeScript sabe que `T` terá uma propriedade `message`.

Como outro exemplo, também poderíamos escrever um tipo chamado `Flatten` que achata tipos de array para seus tipos de elemento, mas os deixa intactos caso contrário:

```
type Flatten<T> = T extends any[] ? T[number] : T;

// Extrai o tipo de elemento.
type Str = Flatten<string[]>;

type Str = string;

// Deixa o tipo intacto.
type Num = Flatten<number>;

type Num = number;
```

Quando `Flatten` recebe um tipo de array, ele usa um acesso indexado com `number` para buscar o tipo de elemento de `string[]`. Caso contrário, ele apenas retorna o tipo que recebeu.

Inferindo Dentro de Tipos Condicionais
Acabamos de nos ver usando tipos condicionais para aplicar restrições e, em seguida, extrair tipos. Isso acaba sendo uma operação tão comum que os tipos condicionais facilitam.

Tipos condicionais nos fornecem uma maneira de inferir de tipos que comparamos no ramo verdadeiro usando a palavra-chave `infer`. Por exemplo, poderíamos ter inferido o tipo de elemento em `Flatten` em vez de buscá-lo "manualmente" com um tipo de acesso indexado:

```
type Flatten<Type> = Type extends Array<infer Item> ? Item : Type;
```

Aqui, usamos a palavra-chave `infer` para introduzir declarativamente uma nova variável de tipo genérico chamada `Item`, em vez de especificar como recuperar o tipo de elemento de `Type` dentro do ramo verdadeiro. Isso nos livra de ter que pensar em como explorar e separar a estrutura dos tipos que nos interessam.

Podemos escrever alguns úteis aliases de tipo auxiliar usando a palavra-chave `infer`. Por exemplo, para casos simples, podemos extrair o tipo de retorno de tipos de função:

```
type GetReturnType<Type> = Type extends (...args: never[]) => infer Return
  ? Return
  : never;

type Num = GetReturnType<() => number>;

type Num = number;

type Str = GetReturnType<(x: string) => string>;

type Str = string;

type Bools = GetReturnType<(a: boolean, b: boolean) => boolean[]>;

type Bools = boolean[];
```

Ao inferir de um tipo com várias assinaturas de chamada (como o tipo de uma função sobrecarregada), as inferências são feitas a partir da última assinatura (que, presumivelmente, é o caso mais permissivo). Não é possível realizar a resolução de sobrecargas com base em uma lista de tipos de argumento.

```
declare function stringOrNum(x: string): number;
declare function stringOrNum(x: number): string;
declare function stringOrNum(x: string | number): string | number;

type T1 = ReturnType<typeof stringOrNum>;

type T1 = string | number;
```

Tipos Condicionais Distributivos

Quando os tipos condicionais atuam em um tipo genérico, eles se tornam distributivos quando recebem um tipo de união. Por exemplo, veja o seguinte:

```
type ToArray<Type> = Type extends any ? Type[] : never;
```

Se inserirmos um tipo de união em `ToArray`, então o tipo condicional será aplicado a cada membro dessa união.

```
type ToArray<Type> = Type extends any ? Type[] : never;

type StrArrOrNumArr = ToArray<string | number>;

type StrArrOrNumArr = string[] | number[];
```

O que acontece aqui é que `ToArray` distribui sobre:

```
  string | number;
```

e mapeia cada tipo membro da união para o que é efetivamente:

```
  ToArray<string> | ToArray<number>;
```

o que nos deixa com:

```
  string[] | number[];
```

Normalmente, a distributividade é o comportamento desejado. Para evitar esse comportamento, você pode cercar cada lado da palavra-chave `extends` com colchetes.

```
type ToArrayNonDist<Type> = [Type] extends [any] ? Type[] : never;

// 'ArrOfStrOrNum' não é mais uma união.
type ArrOfStrOrNum = ToArrayNonDist<string | number>;

type ArrOfStrOrNum = (string | number)[];
```