Tipos Mapeados

Quando você não quer se repetir, às vezes um tipo precisa ser baseado em outro tipo.

Os tipos mapeados são construídos sobre a sintaxe para assinaturas de índice, que são usadas para declarar os tipos de propriedades que não foram declaradas previamente:

```
type ApenasBoolsEHorses = {
  [key: string]: boolean | Horse;
};

const conforma: ApenasBoolsEHorses = {
  del: true,
  rodney: false,
};
```

Um tipo mapeado é um tipo genérico que usa uma união de `ChavesPropriedades` (frequentemente criada via um `keyof`) para iterar através das chaves e criar um tipo:

```
type OpçõesFlags<Type> = {
  [Propriedade in keyof Type]: boolean;
};
```

Neste exemplo, `OpçõesFlags` pegará todas as propriedades do tipo `Type` e mudará seus valores para serem booleanos.

```
type Recursos = {
  modoEscuro: () => void;
  novoPerfilUsuario: () => void;
};

type OpçõesRecursos = OpçõesFlags<Recursos>;

type OpçõesRecursos = {
    modoEscuro: boolean;
    novoPerfilUsuario: boolean;
}
```

Modificadores de Mapeamento

Existem dois modificadores adicionais que podem ser aplicados durante o mapeamento: `readonly` e `?`, que afetam a mutabilidade e a opcionalidade, respectivamente.

Você pode remover ou adicionar esses modificadores prefixando com `-` ou `+`. Se você não adicionar um prefixo, então `+` é assumido.

```
// Remove os atributos 'readonly' das propriedades de um tipo
type CriarMutável<Type> = {
  -readonly [Propriedade in keyof Type]: Type[Propriedade];
};

type ContaBloqueada = {
  readonly id: string;
  readonly nome: string;
};
```

```
type ContaDesbloqueada = CriarMutável<ContaBloqueada>;

type ContaDesbloqueada = {
    id: string;
    nome: string;
}
```

```
// Remove os atributos 'optional' das propriedades de um tipo
type Concreto<Type> = {
  [Propriedade in keyof Type]-?: Type[Propriedade];
};

type TalvezUsuario = {
  id: string;
  nome?: string;
  idade?: number;
};

type Usuario = Concreto<TalvezUsuario>;

type Usuario = {
    id: string;
    nome: string;
    idade: number;
}
```

Remapeamento de Chaves via `as`

No TypeScript 4.1 e posteriores, você pode remapear chaves em tipos mapeados com uma cláusula `as` em um tipo mapeado:

```
type TipoMapeadoComNovasPropriedades<Type> = {
    [Propriedades in keyof Type as NovoTipoChave]: Type[Propriedades]
}
```

Você pode aproveitar recursos como tipos de literais de template para criar novos nomes de propriedades a partir de anteriores:

```
type Getters<Type> = {
    [Propriedade in keyof Type as `get${Capitalize<string & Propriedade>}`]: () =>
Type[Propriedade]
};

interface Pessoa {
    nome: string;
    idade: number;
    localizacao: string;
}

type PessoaPreguiçosa = Getters<Pessoa>;

type PessoaPreguiçosa = {
    getNome: () => string;
    getIdade: () => number;
```

```
    getLocalizacao: () => string;
}
```

Você pode filtrar chaves produzindo `never` via um tipo condicional:

```
// Remove a propriedade 'kind'
type RemoverCampoTipo<Type> = {
    [Propriedade in keyof Type as Exclude<Propriedade, "kind">]: Type[Propriedade]
};

interface Circulo {
    kind: "circle";
    radius: number;
}

type CirculoSemTipo = RemoverCampoTipo<Circulo>;

type CirculoSemTipo = {
    radius: number;
}
```

Você pode mapear sobre uniões arbitrárias, não apenas uniões de `string | number | symbol`, mas uniões de qualquer tipo:

```
type ConfiguracaoEvento<Eventos extends { kind: string }> = {
    [E in Eventos as E["kind"]]: (evento: E) => void;
}

type EventoQuadrado = { kind: "square", x: number, y: number };
type EventoCirculo = { kind: "circle", radius: number };

type Config = ConfiguracaoEvento<EventoQuadrado | EventoCirculo>

type Config = {
    square: (evento: EventoQuadrado) => void;
    circle: (evento: EventoCirculo) => void;
}
```

Exploração Adicional

Os tipos mapeados funcionam bem com outros recursos nesta seção de manipulação de tipos, por exemplo, aqui está um tipo mapeado usando um tipo condicional que retorna `true` ou `false`, dependendo se um objeto tem a propriedade `pii` definida como o literal `true`:

```
type ExtrairPII<Type> = {
  [Propriedade in keyof Type]: Type[Propriedade] extends { pii: true } ? true :
false;
};

type CamposDB = {
  id: { format: "incrementing" };
  nome: { type: string; pii: true };
};

type ObjetosNecessitandoDeletarGDPR = ExtrairPII<CamposDB>;
```

```
type ObjetosNecessitandoDeletarGDPR = {
    id: false;
    nome: true;
}
```