Lekcja 38: Rozstrzygające warunki warunkowe

Zanim przejdziemy do dziedzin typów warunkowych z większą liczbą przykładów, zatrzymajmy się przy jednym typie warunkowym, który właśnie napisaliśmy.

type FetchParams = number

| Customer

| Product

type FetchReturn<Param extends FetchParams> =

Param extends Customer ? Order[] :

Param extends Product ? Order[] : Order

Zapamiętaj metaforę z poprzedniej Lekcji : typy ogólne działają jak funkcje, mają parametry i zwracają dane wyjściowe. Mając to na uwadze, możemy zobaczyć, jak działa ten typ warunkowy, gdy jako argument wstawimy jeden typ.

Powiążmy Param z Customer:

type FetchByCustomer = FetchReturn<Customer>

Zastąp definicją warunku:

type FetchByCustomer =

Customer extends Customer ? Order[] :

Customer extends Product ? Order[] : Order

Sprawdź warunki i uzyskaj wynik.

type FetchByCustomer = Order[]

Możemy przeprowadzić ten sam proces ze wszystkimi innymi kompatybilnymi typami.

Dystrybucja w związkach

Trochę inaczej jest, gdy mijamy typy związków. W większości przypadków typy warunkowe są dystrybuowane w ramach unii podczas tworzenia instancji. Zobaczmy, jak to działa w praktyce. Najpierw tworzymy wystąpienie FetchParam z unią Product i number.

type FetchByProductOrId =

FetchReturn<Product | number>

FetchReturn jest dystrybucyjnym typem warunkowym. Oznacza to, że każdy składnik parametru typu ogólnego jest tworzony z tym samym typem warunkowym. W skrócie: typ warunkowy typu unii jest jak suma typów warunkowych.

type FetchByProductOrId =

(

Product extends Customer ? Order[] :

Product extends Product ? Order[] : Order

) |

(

number extends Customer ? Order[] :

number extends Product ? Order[] : Order

)

Ponownie sprawdzamy warunki, aby uzyskać wynik.

type FetchByProductOrId = Order[] | Order

I to jest nasz oczekiwany wynik!

Świadomość, że typy warunkowe TypeScript działają poprzez dystrybucję, jest niezwykle ważna z kilku powodów.

1. Możemy śledzić każdy typ wejścia do dokładnie jednego typu wyjścia, bez względu na to, w jakiej kombinacji występują.

2. Oznacza to, że w scenariuszu takim jak nasz, w którym chcemy mieć różne typy zwracanych wartości dla różnych typów danych wejściowych, możemy być pewni, że nie zapomnimy o kombinacji. Możliwe kombinacje typów zwracanych to dokładnie możliwe kombinacje typów danych wejściowych.

Mimo że możliwe kombinacje są takie same, związki typu zwracanego usuwają duplikaty i niemożliwe wyniki. Oznacza to, że jeśli dokonamy rozkładu na wszystkie możliwe typy danych wejściowych, w wyniku otrzymamy dwa typy wyjściowe:

type FetchByProductOrId =

FetchReturn<Product | Customer | number>

// Equal to

type FetchByProductOrId =

(

Product extends Customer ? Order[] :

Product extends Product ? Order[] : Order

) |

(

Customer extends Customer ? Order[] :

Customer extends Product ? Order[] : Order

) |

(

number extends Customer ? Order[] :

number extends Product ? Order[] : Order

)

// Equal to

type FetchByProductOrId =

Order[] | Order[] | Order

// Removed redundancies

type FetchByProductOrId =

Order[] | Order

Jest to funkcja, która będzie ważna w dalszej części tej Lekcji.

Nagie typy

Ważnym warunkiem wstępnym dystrybucyjnych typów warunkowych jest to, że parametr typu ogólnego jest typem nagim. Typ nagi to żargon systemu typów i oznacza, że parametr typu jest obecny w takiej postaci, w jakiej jest, bez bycia częścią żadnej innej konstrukcji. Bycie nagim jest najczęstszym przypadkiem w przypadku parametrów typu ogólnego. Wersja non-naked może prowadzić do interesujących efektów ubocznych. Umieśćmy parametr typu w typ krotki.

type FetchReturn<Param extends FetchParams> =

[Param] extends [Customer] ? Order[] :

[Param] extends [Product] ? Order[] : Order

W przypadku powiązań z pojedynczym typem typ warunkowy działa jak poprzednio:

type FetchByCustomer = FetchReturn<Customer>

type FetchByCustomer =

// This condition is still true!

[Customer] extends [Customer] ? Order[] :

[Customer] extends [Product] ? Order[] : Order

type FetchByCustomer = Order[]

Krotka [Param] po utworzeniu wystąpienia z Customer nadal jest podtypem krotki [Customer], więc ten warunek nadal występuje jako Order[]. Kiedy tworzymy wystąpienie Param z typem unii, a ten nie jest dystrybuowany, otrzymujemy następujący wynik:

type FetchByCustomerOrId

= FetchReturn<Customer | number >

type FetchByProductOrId =

// This is false!

[Customer | number] extends [Customer] ? Order[] :

// This is obviously also false

[Customer | number] extends [Product] ? Order[] :

// So we resolve to this

Order

type FetchByProductOrId = Order // Gasp!

[Customer | number], jako szerszy typ, jest nadtypem [Customer] i dlatego nie rozszerza [Customer]. Żaden warunek nie ma zastosowania, a nasz typ warunkowy przechodzi do ostatniej opcji, Order. A to jest fałszywy wynik. Aby ten typ warunkowy był dużo bezpieczniejszy i bardziej poprawny, możemy dodać do niego kolejny warunek, w którym sprawdzimy podtyp number. Ostatnia gałąź warunkowa mówi, never.

type FetchReturn<Param extends FetchParams> =

[Param] extends [Customer] ? Order[] :

[Param] extends [Product] ? Order[] :

[Param] extends [number] ? Order : never

Gwarantuje to, że na pewno otrzymamy poprawną wartość zwracaną, jeśli pracujemy z jednym typem. Typy związków zawsze postanawiają nigdy, co może być dobrym sposobem upewnienia się, że najpierw ograniczymy się do jednego składnika związku.