TypeScript 에서 조건부 타입을 이용한 안전한 코드 작성하기

https://slides.com/woongjae/2020-open-sw-festival

Mark Lee

이 웅재

Lead Software Engineer @ProtoPie

Microsoft MVP

TypeScript Korea User Group Organizer

Marktube (Youtube)

목표

TypeScript 의 Conditional Types 에 대해 이해한다.

TypeScript 에 내장된 Conditional Types 를 알아보자.

나만의 Conditional Types 를 만들어본다.

Conditional Types

A conditional type selects one of two possible types based on a condition expressed as a type relationship test:

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

The type above means when `T` is assignable to `U` the type is `X`, otherwise the type is `Y`.

A conditional type `T extends U ? X : Y` is either *resolved* to `X` or `Y`, or *deferred* because the condition depends on one or more type variables. When `T` or `U` contains type variables, whether to resolve to `X` or `Y`, or to defer, is determined by whether or not the type system has enough information to conclude that `T` is always assignable to `U`.

As an example of some types that are immediately resolved, we can take a look at the following example:

```ts
declare function f<T extends boolean>(x: T): T extends true ? string : number;

// Type is 'string | number'
let x = f(Math.random() < 0.5);
//  ^ = let x: string | number
```

Another example would be the `TypeName` type alias, which uses nested conditional types:

```ts
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 T0 = TypeName<string>;
//   ^ = type T0 = "string"
type T1 = TypeName<"a">;
//   ^ = type T1 = "string"
type T2 = TypeName<true>;
//   ^ = type T2 = "boolean"
type T3 = TypeName<() => void>;
```

On this page

Is this page helpful?

Yes No

Item<T> - T 에 따라 달라지는 container

```
interface StringContainer {
  value: string;
  format(): string;
  split(): string[];
}

interface NumberContainer {
  value: number;
  nearestPrime: number;
  round(): number;
}

type Item1<T> = {
  id: T,
  container: any;
};

const item1: Item1<string> = {
  id: "aaaaaa",
  container: null
};
```

Item<T>

T 가 string 이면 StringContainer, 아니면 NumberContainer

```
type Item2<T> = {
  id: T;
  container: T extends string ? StringContainer : NumberContainer;
};

const item2: Item2<string> = {
  id: 'aaaaaa',
  container: null, // Type 'null' is not assignable to type 'StringContainer'.
};
```

Item<T>

T 가 string 이면 StringContainer

T 가 number 면 NumberContainer

아니면 사용 불가

```
type Item3<T> = {
  id: T extends string | number ? T : never;
  container: T extends string
    ? StringContainer
    : T extends number
    ? NumberContainer
    : never;
};

const item3: Item3<boolean> = {
  id: true, // Type 'boolean' is not assignable to type 'never'.
  container: null, // Type 'null' is not assignable to type 'never'.
};
```

ArrayFilter<T>

```ts
type ArrayFilter<T> = T extends any[] ? T : never;

type StringsOrNumbers = ArrayFilter<string | number | string[] | number[]>;

// 1. string | number | string[] | number[]
// 2. never | never | string[] | number[]
// 3. string[] | number[]
```

Table or Dino

```ts
interface Table {
  id: string;
  chairs: string[];
}

interface Dino {
  id: number;
  legs: number;
}

interface World {
  getItem<T extends string | number>(id: T): T extends string ? Table : Dino;
}

let world: World = null as any;

const dino = world.getItem(10);
const what = world.getItem(true); // Error! Argument of type 'boolean' is not
assignable to parameter of type 'string | number'.ts(2345)
```

Flatten<T>

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


const numbers = [1, 2, 3];
type NumbersArrayFlattened = Flatten<typeof numbers>;
// 1. number[]
// 2. number

const person = {
  name: 'Mark',
  age: 38
};

type SomeObjectFlattened = Flatten<typeof person>;
// 1. keyof T --> "id" | "name"
// 2. T["id" | "name"] --> T["id"] | T["name"] --> number | string

const isMale = true;
type SomeBooleanFlattened = Flatten<typeof isMale>;
// true
```

infer

```
type UnpackPromise<T> = T extends Promise<infer K>[] ? K : any;
const promises = [Promise.resolve('Mark'), Promise.resolve(38)];

type Expected = UnpackPromise<typeof promises>; // string | number
```

함수의 리턴 타입 알아내기 - MyReturnType

```
function plus1(seed: number): number {
  return seed + 1;
}

type MyReturnType<T extends (...args: any) => any> = T extends (
  ...args: any
) => infer R
  ? R
  : any;

type Id = MyReturnType<typeof plus1>;

lookupEntity(plus1(10));

function lookupEntity(id: Id) {
  // query DB for entity by ID
}
```

내장 conditional types (1)

```ts
// type Exclude<T, U> = T extends U ? never : T;
type Excluded = Exclude<string | number, string>; // number - diff

// type Extract<T, U> = T extends U ? T : never;
type Extracted = Extract<string | number, string>; // string - filter

// Pick<T, Exclude<keyof T, K>>; (Mapped Type)
type Picked = Pick<{name: string, age: number}, 'name'>;

// type Omit<T, K extends keyof any> = Pick<T, Exclude<keyof T, K>>;
type Omited = Omit<{name: string, age: number}, 'name'>;

// type NonNullable<T> = T extends null | undefined ? never : T;
type NonNullabled = NonNullable<string | number | null | undefined>;
```

내장 conditional types (2)

```
/*
type ReturnType<T extends (...args: any) => any> = T extends (
  ...args: any
) => infer R
  ? R
  : any;
*/

/*
type Parameters<T extends (...args: any) => any> = T extends (
  ...args: infer P
) => any
  ? P
  : never;
*/
type MyParameters = Parameters<(name: string, age: number) => void>; //
[name: string, age: number]
```

Function 인 프로퍼티 찾기

```
type FunctionPropertyNames<T> = {
  [K in keyof T]: T[K] extends Function ? K : never;
}[keyof T];
type FunctionProperties<T> = Pick<T, FunctionPropertyNames<T>>;

type NonFunctionPropertyNames<T> = {
  [K in keyof T]: T[K] extends Function ? never : K;
}[keyof T];
type NonFunctionProperties<T> = Pick<T, NonFunctionPropertyNames<T>>;

interface Person {
  id: number;
  name: string;
  hello(message: string): void;
}

type T1 = FunctionPropertyNames<Person>;
type T2 = NonFunctionPropertyNames<Person>;
type T3 = FunctionProperties<Person>;
type T4 = NonFunctionProperties<Person>;
```

내장 conditional types (3)

```
interface Constructor {
  new (name: string, age: number): string;
}

/*
type ConstructorParameters<
  T extends new (...args: any) => any
> = T extends new (...args: infer P) => any ? P : never;
*/

type MyConstructorParameters = ConstructorParameters<Constructor>; // [name:
string, age: number]

/*
type InstanceType<T extends new (...args: any) => any> = T extends new (
  ...args: any
) => infer R
  ? R
  : any;
*/

type MyInstanceType = InstanceType<Constructor>; // string
```

DeepReadonly<T>

```
type DeepReadonly<T> = T extends (infer E)[]
  ? ReadonlyArray<DeepReadonlyObject<E>>
  : T extends object
  ? DeepReadonlyObject<T>
  : T;

type DeepReadonlyObject<T> = { readonly [K in keyof T]: DeepReadonly<T[K]> };

type IDeepReadonlyRootState = DeepReadonly<IRootState>;
let state2: IDeepReadonlyRootState = {} as IDeepReadonlyRootState;
const book2 = state2.book.books[0];
book2.title = 'new'; // error! Cannot assign to 'title' because it is a read-
only property.
```