Extract

This lesson explains the extract mapped type.

We'll cover the following

Description of `Extract`#

The next mapped type is `Extract`. It allows us to extract from a set of types the one that is common in another type. The following code shows that the first generic argument has three variables (`string`, `string[]`, and `number`) and the second generic type is `unknown[]`, which means any kind of array. `Extract` returns `string[] | number[]` as a type because they are the only two that fulfill the conditions of an array.

```
type OnlyArrayType = Extract<string | string[] | number[], unknown[]>;

const var1: OnlyArrayType = ["Element1"];
const var2: OnlyArrayType = [1];
// const var3: OnlyArrayType = "No";
```

Example#

Although the example is theoretically sound, it is hard to grasp the modification. To understand better, let us dive into something more visual. Let's have two different interfaces that have one or more intersecting members. In our example, the member is `name: string`.

```
interface Animal {
    name: string;
    sound: string;
}
interface Human {
    name: string;
    nickname: string;
}
```

Let's create a function that accepts common members of a defined set of types; for example, the common members of `Animal` and `Human`. To identify the common member, `Extract` comes to the rescue. The following code is identical to `type LivingThing = "name"` which is the only member that intersects both interfaces.

```
type LivingThing = Extract<keyof Animal, keyof Human>;
```

With the combination of `Record`, we can dynamically build a new type that is the intersection of both.

```
type LivingThing = Extract<keyof Animal, keyof Human>;
function sayMyName(who: Record<LivingThing, string>): void {
    console.log(who.name);
}
```

The result is that you can create a function that expects a type that does not exist but is common to two types.

```
interface Animal {
    name: string;

    sound: string;
}
interface Human {
    name: string;
    nickname: string;
}

type LivingThing = Extract<keyof Animal, keyof Human>;
function sayMyName(who: Record<LivingThing, string>): void {
    console.log(who.name);
}
const animal: Animal = { name: "Lion", sound: "Rawwwhhh" };
const human: Human = { name: "Jacob", nickname: "Jaco-bee" };
sayMyName(animal);
sayMyName(human);
```

Advantage#

Furthermore, if the two types evolve, the function will remain type-safe. For example, if both types add a new string type member, the function will allow access to the member. See **line 14** below:

```
interface Animal {
    name: string;
    gender: string;
    sound: string;
}
interface Human {
    name: string;
    gender: string;
    nickname: string;
}

type LivingThing = Extract<keyof Animal, keyof Human>;
function sayMyName(who: Record<LivingThing, string>): void {
    console.log(who.name +" is of type " + who.gender);
}
const animal: Animal = { name: "Lion", sound: "Rawwwhhh", gender: "Male" };
const human: Human = { name: "Jacob", nickname: "Jaco-bee", gender: "Boy" };
sayMyName(animal);
sayMyName(human);
```

The `Extract` mapped type is not as straightforward as the previous ones discussed. It is mainly used in the conjunction of other mapped types as well as with conditional types for advanced scenarios. The examples presented in this lesson demonstrate the concept of `Extract`. A keen developer could extract a common interface between the two types and achieve a better result by explicitly extracting and naming this interface. Nevertheless, `Extract` plays a paramount role in cases where a function is not aware of the business logic and needs to manipulate a generic type.

Back
Record

Next
Exclude

Mark as Completed

Report an Issue