Mashup 11th Node Team

10mins seminar @2021 July 31

Unboxing and Infer

Extends Keyword

Function Argument

Utility Type Composition

Variadic Tuple

Chainable with Recursive Type

Template Literal Type

Unboxing and Infer

Extends Keyword

Function Argument

Utility Type Composition

Variadic Tuple

Chainable with Recursive Type

Template Literal Type

```
type Res = Promise<{ data: string; status: number }>;

type Await<T> = T extends Promise<infer K> ? K : never;
type Await<T> = /* complete here */

type Result = Await<Res>;
// type Result = {
//   data: string;
//   status: number;
// }
```

```
type Await<T> = T extends Promise<infer K> ? K : never;
```

> 외부 라이브러리들은 inner type을 export하지 않을 때가 있는데
>
> 이를 직접 하나하나 정의해주는건 라이브러리 스펙이 바뀌었을 때 대응할 수 없다

다음과 같이 외부 라이브러리에서 직접 타입을 뽑아줄 수 있다

```
import * as fs from 'fs/promises';

type File = Await<ReturnType<typeof fs.readFile>>;
//   ^^^^ = string | Buffer
```

다음과 같이 외부 라이브러리에서 직접 타입을 뽑아줄 수 있다

```
import * as fs from 'fs/promises';

type File = Await<ReturnType<typeof fs.readFile>>;
//   ^^^^ = string | Buffer
```

제너릭으로 감싸진 타입이은 무엇이든지 뽑을 수 있다

```
type ElementTypes<A> = A extends Array<infer I> ? I : never;

const arr = ['string', 0, true];

type ElementOfArray = ElementTypes<typeof arr>;
//   ^^^^^^^^^^^^^^ = string | number | boolean;
```

Q. 다음 코드에서 에러를 찾아보세요

```
type Card = {
  usedAmount: number;
  type: string;
};

function toCreditCard<C extends Card>(c: C): C {
  assert.isCard(c);
  return {
    usedAmount: c.usedAmount,
    type: 'credit',
  };
}
```

```typescript
type Card = {
  usedAmount: number;
  type: string;
};

function toCreditCard<C extends Card>(c: C): C {
  assert.isCard(c);
  return {
    usedAmount: c.usedAmount,
    type: 'credit',
  };
}
```

```
> Type '{ usedAmount: number; type: string; }' is not assignable to type 'C'.
'{ usedAmount: number; type: string; }' is assignable to the constraint of type 'C',
 but 'C' could be instantiated with a different subtype of constraint 'Card'.
```

```
type Card = {
  usedAmount: number;
  type: string;
};

function toCreditCard<C extends Card>(c: C): C {
  assert.isCard(c);
  return {
    usedAmount: c.usedAmount,
    type: 'credit',
  };
}
```

> **C는 Card의 Subtype이므로, Card에 없는 property를 가질 가능성이 있다.**
>
> **따라서 Card literal 형태의 리턴타입을 허용할 수 없다.**

```
type Card = {
  usedAmount: number;
  type: string;
};

function toCreditCard<C extends Card>(c: C): C {
  assert.isCard(c);
  return ...c,{
    usedAmount: c.usedAmount,
    type: 'credit',
  };
}
```

```typescript
type LookUp = /** complete here */

type CardCommon = {
  name: string;
  number: `${string}-${string}-${string}-${string}`;
  expiredAt: Date;
};

type HanaCard = { type: 'visa' } & CardCommon;
type WooriCard = { type: 'master' } & CardCommon;
type ShinhanCard = { type: 'amex' } & CardCommon;
type SamsungCard = { type: 'amex' } & CardCommon;

type Card = HanaCard | WooriCard | ShinhanCard | SamsungCard;

type AmexCards = LookUp<Card, 'amex'>;
//    ^^^^^^^^^ = ShinhanCard | SamsungCard;
```

```typescript
type LookUp<U, T> = U extends { type: T } ? U : never;

type CardCommon = {
  name: string;
  number: `${string}-${string}-${string}-${string}`;
  expiredAt: Date;
};

type HanaCard = { type: 'visa' } & CardCommon;
type WooriCard = { type: 'master' } & CardCommon;
type ShinhanCard = { type: 'amex' } & CardCommon;
type SamsungCard = { type: 'amex' } & CardCommon;

type Card = HanaCard | WooriCard | ShinhanCard | SamsungCard;

type AmexCards = LookUp<Card, 'amex'>;
//   ^^^^^^^^^ = ShinhanCard | SamsungCard;
```

```
// 음악이 앨범에 속하는지 판단하는 함수
declare function isMusicInAlbum(music: Music, album: Album): boolean;

type WithOption = /** complete here */

// 앨범에 속하지 않을 때 throw할 수 있는 옵션인자를 추가로 받을 수 있다.
type ThrowableIsMusicInAlbum = WithOption<typeof isMusicInAlbum, { throw: boolean }>;
//   ^^^^ = (o: { throw: boolean }, music: Music, album: Album) => boolean;
// 음악이 앨범에 속하는지 판단하는 함수
declare function isMusicInAlbum(music: Music, album: Album): boolean;

type WithOption<F extends (...args: any) => any, O> = (
  option: O,
  ...args: Parameters<F>
) => ReturnType<F>;

// 앨범에 속하지 않을 때 throw할 수 있는 옵션인자를 추가로 받을 수 있다.
type ThrowableIsMusicInAlbum = WithOption<typeof isMusicInAlbum, { throw: boolean }>;
//   ^^^^ = (o: { throw: boolean }, music: Music, album: Album) => boolean;
```

```typescript
// 음악이 앨범에 속하는지 판단하는 함수
declare function isMusicInAlbum(music: Music, album: Album): boolean;

type WithOption<F, A> = F extends (...args: infer Args) => infer Return
  ? (x: A, ...args: Args) => Return
  : never;


// 앨범에 속하지 않을 때 throw할 수 있는 옵션인자를 추가로 받을 수 있다.
type ThrowableIsMusicInAlbum = WithOption<typeof isMusicInAlbum, { throw: boolean }>;
//   ^^^^ = (o: { throw: boolean }, music: Music, album: Album) => boolean;
```

Unboxing and Infer

Extends Keyword

Function Argument

Utility Type Composition

Variadic Tuple

Chainable with Recursive Type

Template Literal Type

```typescript
type UserInfo = {
  name?: string;
  birth?: string;
  gender?: string;
  avatar?: string;
  age?: number;
  updatedAt?: Date;
  createdAt?: Date;
};

type Ensured = /** complete here */

type UserNameAndAge = Ensured<UserInfo, 'name' | 'age'>;
//   ^^^^^^^^^^^^^^ = { name: string; age: number; }
```

```
type UserInfo = {
  name?: string;
  birth?: string;
  gender?: string;
  avatar?: string;
  age?: number;
  updatedAt?: Date;
  createdAt?: Date;
};

type Ensured<T, K extends keyof T> = Pick<Required<T>, K>;

type UserNameAndAge = Ensured<UserInfo, 'name' | 'age'>;
//   ^^^^^^^^^^^^^^ = { name: string; age: number; }
```

```
type UserInfo = {
  name?: string;
  birth?: string;
  gender?: string;
  avatar?: string;
  age?: number;
  updatedAt?: Date;
  createdAt?: Date;
};

type Ensured<T, K extends keyof T> = Pick<Required<T>, K>;

type UserNameAndAge = Ensured<UserInfo, 'name' | 'age'>;
//   ^^^^^^^^^^^^^^ = { name: string; age: number; }
```

> **빌트인 유틸리티 타입을 간단하게 조합하면 다양한 유틸리티 타입을 만들 수 있다**
>
> **Pick, Required, Parameters, Extract, Omit, ReturnType, ...**

Unboxing and Infer

Extends Keyword

Function Argument

Utility Type Composition

Variadic Tuple

Chainable with Recursive Type

Template Literal Type

Unboxing and Infer

Extends Keyword

Function Argument

Utility Type Composition

Variadic Tuple

Chainable with Recursive Type

Template Literal Type

```
type NonEmptyArray<T> = /** complete here */

const a: NonEmptyArray<string> = []

const b: NonEmptyArray<string> = ['toss']
```

**> Type '[]' is not assignable to type 'NonEmptyArray<string>'.
Source has 0 element(s) but target requires 1.**

```
type NonEmptyArray<T> = [T, ...T[]];

const a: NonEmptyArray<string> = []

const b: NonEmptyArray<string> = ['toss']
```

**> Type '[]' is not assignable to type 'NonEmptyArray<string>'.
Source has 0 element(s) but target requires 1.**

```
type Repeat2<T extends readonly any[]> = [...T, ...T];

// type SNSN = [string, number, string, number]
type SNSN = Repeat2<[string, number]>;
// type BSNSNB = [boolean, string, number, string, number, boolean]
type BSNSNB = [boolean, ...SNSN, boolean]
```

> **Variadic tuple 타입은 Array를 typesafe하게 활용할 수 있도록 도와준다**

```
type Chainable = /** complete here */

declare const httpOptionBuilder: Chainable;

enum HttpMethod {
  GET = 'GET',
  POST = 'POST',
}
const httpOption: HttpOption = httpOptionBuilder
  .option('host', 'https://mash-up.it')
  .option('method', HttpMethod.POST)
  .option('header', { 'x-mash-up-team': 'node', cookie: 1234 })
  .build(); //                                          ^^^^
  //                 Type 'number' is not assignable to type 'string'

type HttpOption = {
  host: string;
  method: HttpMethod;
  header: {
    'x-mash-up-team': string;
    cookie: string;
  };
};
```

```
type Chainable<T = {}> = {
  option<K extends string, V>(key: K, value: V): Chainable<Omit<T, K> & Record<K, V>>;
  build(): T;
};
```

```
declare const httpOptionBuilder: Chainable;

enum HttpMethod {
  GET = 'GET',
  POST = 'POST',
}
const httpOption: HttpOption = httpOptionBuilder
  .option('host', 'https://mash-up.it')
  .option('method', HttpMethod.POST)
  .option('header', { 'x-mash-up-team': 'node', cookie: 1234 })
  .build(); //                                          ^^^^
  //                 Type 'number' is not assignable to type 'string'
```

```
type Chainable<T = {}> = {
  option<K extends string, V>(key: K, value: V): Chainable<Omit<T, K> & Record<K, V>>;
  build(): T;
};
```

NOTE: 타입 Recursion은 최대 Depth가 44로 정해져있다.

44개 이상의 option을 추가하면 타입추론이 동작하지 않는다.

```
type ExtractRouteParams = /** complete here */

type P = ExtractRouteParams<'/posts/:postId'>;
//    ^ = { postId: string }
type C = ExtractRouteParams<'/posts/:postId/comments/:commentId'>;
//    ^ = { postId: string; commentId: string }
```

```typescript
type ExtractRouteParams<T extends string> =
  string extends T
  ? Record<string, string>
  : T extends `${infer Start}:${infer Param}/${infer Rest}`
  ? {[k in Param | keyof ExtractRouteParams<Rest>]: string}
  : T extends `${infer Start}:${infer Param}`
  ? {[k in Param]: string}
  : {};

type P = ExtractRouteParams<'/posts/:postId'>;
//    ^ = { postId: string }
type C = ExtractRouteParams<'/posts/:postId/comments/:commentId'>;
//    ^ = { postId: string; commentId: string }
```

```
type ParseSingle<S extends string> = S extends `${infer K}=${infer T}`
  ? { [k in K]: T extends 'string' ? string : number }
  : S extends `${infer K}`
  ? { [k in K]: true }
  : {};
type Flatten<T> = { [k in keyof T]: T[k] };
type Tup<A, B> = A extends B ? (B extends A ? A : [A, B]) : [A, B];
type List<H, T> = T extends any[] ? [H, ...T] : Tup<H, T>;
type Compose<T, R> = keyof T extends keyof R
  ? Flatten<{ [k in keyof T]: List<T[k], R[k]> } & Omit<R, keyof T>>
  : Flatten<T & R>;
type ParseQueryString<S extends string> = S extends ''
  ? {}
  : S extends `${infer P}&${infer Rest}`
  ? Compose<ParseSingle<P>, ParseQueryString<Rest>>
  : ParseSingle<S>;

type MovieLookupQueryTemplate = `page=50&limit=100&category=comedy`;
type MovieLookupQuery = ParseQueryString<MovieLookupQueryTemplate>;
//   ^^^^^^^^^^^^^^^^ = { page: number;  limit: number; category: string; }
```

Mashup 11th Node Team

10mins seminar @2021 July 31