[TS] 7. Typescript's Type System

by jbee

Table of Contents

TypeScript의 Typing Checking System

`TypeScript` 에서의 Type System에 대한 이해를 하기 전, 기존 프로그래밍 언어의 큰 두 축인 정적 언어와 동적 언어에 대한 정의를 다시 한 번 살펴볼 필요가 있습니다.

정적언어 (Static Language)

- 변수(variables) 또는 함수(function)의 `Type` 을 **미리** 지정해야 한다.
- 컴파일되는 시점에 Type Check를 수행한다.

동적 언어 (Dynamic Language)

- 변수(variables) 또는 함수(function)의 `Type`을 지정하지 않는다.
- Type Check는 런타임(runtime) 환경에서나 알 수 있다.

Duck Typing

덕 타이핑(Duck Typing)이라고 많이 들어보셨을 텐데요, 현재 이 덕 타이핑 체계를 기반으로 동적 언어에 타입을 추론하는 언어는 GoLang과 Python 등이 있습니다. 하지만 TypeScript는 이 덕타이핑과는 조금 다른 체계로 Typing을 하고 있습니다.

> 덕 타이핑에 대한 보다 자세한 내용은 여기를 참고해주세요.

Structural typing

TypeScript는 Structural typing (구조적 타이핑)을 기반으로 타입 시스템을 갖추고 있습니다. 구조적 타이핑이란, `멤버`에 따라 타입을 **연관짓는** 방법을 말합니다. 구조적 타이핑과 반대인 방법으로 `nominal typing`이 있습니다. 우리가 알고 있는 일반적인 정적 언어인 C#, Java는 이 `nominal typing` 방식으로 type checking이 이루어집니다.

_Official Document_에 나온 예제를 통해 설명드립니다.

```
interface Named {
  name: string;
}

class Person {
  name: string;
}

let p: Named;
p = new Person(); // Ok, because of structural typing
```

위 코드를 C# 또는 Java의 문법에 맞게 변경한다면 동작하지 않는 잘못된 코드가 됩니다. 하지만 TypeScript에서는 정상적으로 동작합니다. `Named` 와 `Person` 두 가지는 오로지 `name` 이라는 프로퍼티(or 멤버)만 갖고 있는 타입이므로 서로 `compatibility` 하다고 볼 수 있습니다. 때문에 위 코드는 문제되지 않습니다. (compatibility에 대해서는 뒤에서 알아봅니다.)

Type Inference

'누가봐도 이 변수는 이 타입이다.'라는 것에 대해 TypeScript가 지원해주는 것을 **타입 추론** 이라고 합니다.

```
let name = `jbee`;
```

`name` 이라는 변수는 `string` 타입의 변수가 됩니다. 그렇기 때문에 굳이 `: string` 이라고 타입을 지정해주지 않아도 다음과 같이 에러가 발생합니다.

```
name = 1; // Error: Type '1' is not assignable to type 's
```

그렇다면 다음과 같은 경우에는 어떻게 추론될까요?

```
const mixedArr = [0, 1, `jbee`];
```

`number`에 해당되는 value와 `string`에 해당하는 value가 공존하기 때문에 위 코드에서 `mixedArr`은 `(number | string)[]`의 타입을 갖게 됩니다. 이렇게 여러 타입이 공존하는 경우에 추론하여 지정되는 타입을 Best common type이라고 합니다.

Type Assertion

이 변수의 타입은 분명 `A` 인데 TypeScript에서 보수적으로 처리하여 에러를 발생시키는 경우가 있습니다. 이럴 경우 해당 변수를 `A` 라고 명시하여 에러를 사라지게 할 수 있습니다.

```
export type Todo = {
  id: number;
  text: string;
  completed: boolean;
};

export type Todos = Todo[];
```

위와 같이 `Todo` 타입과 `Todos` 타입을 지정한 상황이라고 했을 때를 예로 들어보겠습니다.

```
const initialState: Todos = [
  {
    id: 0,
    text: 'Study RxJS',
    completed: false,
  }
];
```

위 코드에서 `Todos` 에 해당하는 타입을 제대로 지정해줬지만 뭔가 아쉬움이 남을 수 있는데요, 이 때 두 가지 방법을 사용할 수 있습니다.

```
const initialState: Todos = [
  <TODO>{
    id: 0,
    text: 'Study RxJS',
    completed: false,
  }
];
```

위 코드처럼 `<>` 을 사용하거나

```
const initialState: Todos = [
  {
    id: 0,
    text: 'Study RxJS',
    completed: false,
  } as Todo
];
```

위 코드처럼 `as` 키워드를 사용할 수 있습니다. 두 가지 모두 동일하지만 `tsx` 와 함께 사용하기 위해서는 `as` 키워드를 사용하는 것이 좋습니다.

> [!] 이 Type Assertion은 Type Casting과는 다릅니다. 자세한 내용은 DailyEngineering - 타입 추론과 타입 단언을 참고해주세요!

Type Guards

자바스크립트에서는 `typeof` 또는 `instanceof`와 같은 오퍼레이터가 타입을 확인해주는 역할을 했었습니다.

```
if (typeof this.state[key] !== typeof newData) {
    return ;
}
```

하지만 이 방법은 런타임에서 타입 체크를 수행하게 됩니다. 따라서 컴파일 시점에서는 올바른 타입인지 알 수 없습니다. TypeScript에서는 컴파일 시점에서 타입 체크를 수행할 수 있도록 `Type Guard`를 지원합니다.

`typeof`, `instanceof`

TypeScript에서도 마찬가지로 `typeof`와 `instanceof` 오퍼레이터를 지원합니다.

```
function setNumberOrString(x: number | string) {
  if (typeof x === 'string') {
    console.log(x.subtr(1)); // Error
    console.log(x.substr(1)); // OK
  } else {
    console.log(typof x); // number
  }
  x.substr(1); // Error
}
```

위 코드에서 `if-block` 내에서의 `x` 변수의 타입은 `string` 일 수 밖에 없다는 것을 컴파일 시점에 체크하여 transpiler가 Error를 발생시키는 경우입니다. 이 예제와 비슷한 방법으로 `instanceof` 오퍼레이터를 사용할 수 있습니다. `instanceof`는 클래스를 기반으로 생성된 인스턴스의 타입을 판단하는데 사용됩니다.

```
class Pet {
  name = 123;
  common = '123';
}

class Basket {
  size = 123;
  common = '123';
}
```

```
function create(arg: Pet | Basket) {
  if (arg instanceof Pet) {
    console.log(arg.name); // OK
    console.log(arg.size); // Error!
  }
  if (arg instanceof Basket) {
    console.log(arg.name); // Error!
    console.log(arg.size); // OK
  }

  console.log(arg.common); // OK
  console.log(arg.name); // Error!
  console.log(arg.size); // Error!
}
```

`typeof` 와 마찬가지로 `instanceof` 로 필터링 된 block 내부에서 Type checking이 이루어집니다.

in

```
interface A {
  x: number;
}
interface B {
  y: string;
}

function execute(q: A | B) {
  if ('x' in q) {
    // q: A
  } else {
    // q: B
  }
}
```

`A` 또는 `B` 를 유니온 타입으로 받을 수 있는 `execute` 함수 내에서 각각에 대해 다른 처리를 할 경우, `in` 이라는 오퍼레이터를 사용할 수 있습니다. 해당 오퍼레이터는 check하고자 하는 타입에 해당 프로퍼티가 존재하는지의 유무를 판단할 수 있습니다.

`.kind` Literal Type Guard

사용자에 의해 `type` 으로 정의된 타입에 대해서, 즉 TypeScript 내부에서 지원하는 primitive type이 아닌 **사용자 정의 타입에 대해서 타입 검사를 수행할 때**, `.kind` 를 사용할 수 있습니다.

```ts
type Foo = {
  kind: 'foo', // Literal type
  foo: number
}
type Bar = {
  kind: 'bar', // Literal type
  bar: number
}

function execute(arg: Foo | Bar) {
  if (arg.kind === 'foo') {
    console.log(arg.foo); // OK
    console.log(arg.bar); // Error!
  }
}
```

위 코드에서 처럼 `type` 키워드를 사용하여 별도 타입을 지정할 때, `kind` 라는 프로퍼티를 추가하여 타입 검사를 수행할 수 있습니다.

User Defined Type Guards

메소드를 별도로 분리하여 Type Guard를 지정할 수 있습니다. 아까 지정한 interface A로 만들어보겠습니다.

```
interface A {
  x: number;
}
// Define Type Guard
function isA(arg: any): arg is A {
  return arg.x !== undefined;
}
```

`arg is A`라는 타입으로 `isA` 메소드가 Type Guard의 역할을 수행한다는 것을 명시할 수 있습니다. if 내부에 들어가는 로직을 별도로 추출하여 보다 가독서이 좋은 코드를 작성할 수 있습니다.

Type Compatibility

한국어로 번역하게 되면 **타입 호환성** 정도로 할 수 있겠는데요, TypeScript는 위에서 언급했듯이 Structural subtyping을 기반으로 Type checking을 하기 때문에 이를 기반으로 타입 간의 호환성을 고려할 수 있습니다.

1. Comparing two Objects

```
let x = {name: `Jbee`};
let y = {name: `James`, age: 34};

x = y; // OK!
y = x; // Error!
```

위 코드에서 `x` 는 `{name: string}` 타입으로 추론되며, `y` 는 `{name: string, age: number}` 로 추론됩니다. 이 경우 `x = y` 는 `y` 에 `name` 이라는 속성이 있으므로 가능하지만 `y = x` 의 경우, `x` 에는 `age` 라는 속성이 없으므로 에러가 발생합니다.

2. Comparing two functions

```
let x = (a: number) => 0;
let y = (b: number, s: string) => 0;

y = x; // OK
x = y; // Error
```

함수일 경우에는 객체인 경우와 조금 다른 것을 볼 수 있습니다. 위 코드에서 두 `x`, `y` 함수는 parameter 만 다르게 정의되어 있습니다. 이 경우, `x` 함수에 전달할 수 있는 parameter의 경우의 수가 y에 모두 해당하므로 `y = x` 가 정상적으로 동작합니다. 하지만 그 반대인 `x = y` 는 `y` 함수에 전달할 수 있는 parameter를 `x` 가 모두 포용할 수 없으므로 에러가 발생합니다.

```
let x = () => ({name: `Jbee`});
let y = () => ({name: `James`, age: 34});

x = y; // OK!
y = x; // Error!
```

이번에는 return value의 type이 다른 경우입니다. 이 경우에는 함수의 경우를 따르지 않고 객체인 경우를 따르게 됩니다.

마무리

TypeScript의 단순한 문법을 조금 넘어서 어떻게 Type Checking이 이루어지는지 살펴봤습니다.
감사합니다.

Reference

- TypeScript Official Document - Type Inference
- TypeScript Official Document - Type Compatibility
- Golang으로 만나보는 duck typing
- Type Systems: Structural vs Nominal typing explained
- TypeScript Deep dive - Type Guard