EFFECTIVE TYPESCRIPT

Any

240608 두선아

Table of Contents

타입 시스템은 선택적 (OPTION) 이고 점진적 (GRADUAL) 이기 때문에 정적이면서도 동적인 특성을 가짐 동시에 가집니다.

Optional Typing

선택적 타입 시스템

프로그램의 일부에만 타입 시스템을 적용할 수 있습니다.
점진적 마이그레이션을 할 수 있는 이유가
바로 부분적으로 타입 체크를 비활성화시켜주는
ANY 타입이 있기 때문입니다.

Gradual Typing

점진적 타입 시스템

프로그램의 일부에만 타입 시스템을 적용할 수 있습니다.
점진적 마이그레이션을 할 수 있는 이유가
바로 부분적으로 타입 체크를 비활성화시켜주는
ANY 타입이 있기 때문입니다.

Dynamic Typing

동적 특성 / 동적 타이핑

타입을 명시하지 않은 변수나 함수는 동적으로 타입이 결정됩니다.
즉, 런타임 시에 변수의 타입이 결정되며
다양한 타입의 값을 가질 수 있습니다.

타입을 명시하지 않은 부분은 런타임에 동적으로 타입을 결정합니다.

Static Typing

정적 특성 / 정적 타이핑

타입을 명시한 변수나 함수는 정적으로 타입이 결정됩니다.
즉, 컴파일 시에 타입이 고정되며, 해당 타입의 값만을 가질 수 있다.

컴파일 타임에 타입 오류를 발견하고 수정합니다.
타입을 명시함으로써 IDE에서의 자동 완성을 제공하고,
코드 리팩토링, 타입 안전성 등의 혜택을 받을 수 있습니다.

38.

any 타입은 가능한 한 좁은 범위에서만 사용하기

Use the Narrowest Possible Scope for any Types

```ts
function f1() {
  const x: any = expressionReturningFoo(); // ❌ 🤭
  processBar(x);
}
function f2() {
  const x = expressionReturningFoo(); // better.
  processBar(x as any);
  // 함수의 매개변수에만 사용된 표현식이므로 다른 코드에는 영향을 미치지 않기 때문
  // @ts-ignore, @ts-expect-error를 사용해서 any를 사용하지 않고 오류 제거도 가능. (런타임 오류 가능성)
}
```

function & any

함수와 any

함수의 반환 타입은?
any가 반환되어 함수 바깥에
영향을 미치는 것을 방지하기 위해,
항상 반환 타입을 명시하는 것이 좋습니다.

```ts
const config: Config = {
  a: 1,
  b: 2,
  c: {
    key: value,
  },
} as any; // ❌

const config: Config = {
  a: 1,
  b: 2, // 다른 속성은 타입 체크
  c: {
    key: value as any, // 필요한 속성의 값만 any
  },
};
```

object & any

객체 속성과 any

객체 속성의 any 타입은?
전체 객체에 any를 사용하는 것이 아니라,
필요한 속성의 값만 적용해서
다른 속성에 영향을 미치지 않도록 한다.

Use the Narrowest Possible Scope for any Types

의도치 않은 타입 안정성의 손실을 피하기 위해서
any를 최대한 좁게 사용하기

당신이 **any**를 반환한다면...
함수를 호출한 곳에서 타입 안정성을
아무도 모르게 조용히 잃게 됩니다

39.

any를 구체적으로 변형해서 사용하기

Prefer More Precise Variants of any to Plain any

any는?

자바스크립트에서 표현할 수 있는 모든 값의 범위입니다.

- 숫자, 문자열, 배열, 객체, 정규식, 함수, 클래스, DOM 엘리먼트, null, undefined

따라서 값을 표현할 때, **any**보다는 더 구체적으로 표현할 수 있는 타입이 있습니다.

any[]

any 타입의 매개변수는 모든 값이 될 수 있습니다.

```
function getLengthBad(array: any) {
  // ❌
  return array.length;
}
getLengthBad(/123/);
getLengthBad(null);
```

이중배열은? any[][]입니다.

any[] 타입은 모든 값이 될 수 있는 요소의 배열을 표현한 타입입니다.

```
function getLength(array: any[]) {
  return array.length; // This is better !
}
getLength(/123/); // 오류
//        ~~~~~
// Argument of type 'RegExp' is not assignable to parameter of type 'any[]'.
getLength(null); // 오류
//        ~~~~
// Argument of type 'null' is not assignable to parameter of type 'any[]'.
```

- 함수 내의 array.length 타입을 체크합니다.
- 함수의 반환 타입이 any 대신 number입니다.
- 함수가 호출될 때, 매개변수가 배열인지 체크합니다.

any 객체

값을 알 수 없는 객체라면?

```
function hasAKeyThatEndsWithZ(o: Record<string, any>) {
  for (const key in o) {
    if (key.endsWith("z")) {
      console.log(key, o[key]); // key를 통해 속성에 접근
      return true;
    }
  }
  return false;
}
```

{[key: string]: any} && Record<string, any>

구체적인 키와 값의 타입을 동적으로 정의하는 경우입니다.

```
function hasAKeyThatEndsWithZ(o: object) {
  for (const key in o) {
    if (key.endsWith("z")) {
      console.log(key, o[key]);
      //                ~~~~~~ Element implicitly has an 'any' type
      //                       because type '{}' has no index signature
      // {} 형식에 인덱스 시그니처가 없으므로 요소에 암시적인 any가 있음
      return true;
    }
  }
  return false;
}
```

object

비기본형 non-primitive 타입을 포함하는 object 타입입니다.
키를 열거할 수는 있지만, 속성에 접근할 수 없습니다.
객체의 구조나 프로퍼티에 대해 타입 검사를 할 수 없는데,
이유는 구체적인 키-값 쌍을 나타내지 않았기 때문입니다.

함수의 타입과 any

타입을 알 수 없는 함수라면?

```
type Fn0 = () => any; // 매개변수 없이 호출 가능한 모든 함수
type Fn1 = (arg: any) => any; // 매개변수 1개
type FnN = (...args: any[]) => any; // 모든 개수의 매개변수 === Function
// args가 배열 형태임을 알 수 있다.
const numArgsBad = (...args: any) => args.length; // any를 반환한다
const numArgsGood = (...args: any[]) => args.length; // nubmer를 반환한다
```

Prefer More Precise Variants of any to Plain any

🤔

any를 사용할 때
모든 자바스크립트 값이
정말로 허용되어야 하는 지 생각하기

🤔..?

그냥 **any**보다 더 정밀한 형태를
모델링할 수 있습니다

40.

함수 안으로 타입 단언문 감추기

Hide Unsafe Type Assertions in Well-Typed Functions

함수 안에서 any를 사용하는 경우

외부 타입 정의는 간단하지만, 함수 내부 로직이 복잡한 경우
내부에 타입 단언을 사용하고
외부로 드러나는 타입 정의를 정확히 명시합니다.
프로젝트에 타입 단언문이 드러나지 않도록 함수 내부에 감춰야 합니다.

예시A: 캐시 래퍼 함수

함수 내부에 any가 많지만
cacheLast의 타입 정의에는 any가 없기 때문에
cacheLast를 호출하는 쪽에서는
any가 사용되었는지 알지 못합니다

any를 사용했지만
cacheLast를 사용하는 쪽에서
타입 안전성을 유지합니다

```ts
declare function cacheLast<T extends Function>(fn: T): T;
declare function shallowEqual(a: any, b: any): boolean;

// 구현체
// 원본 함수가 객체처럼 속성 값을 가지고 있다면 타입이 달라진다
// (연속적으로 호출할 때 this 값이 동일한 지 체크하지 않음)
function cacheLast<T extends Function>(fn: T): T {
  let lastArgs: any[] | null = null;
  let lastResult: any;

  return function (...args: any[]) {
    if (!lastArgs || !shallowEqual(lastArgs, args)) {
      lastResult = fn(...args); // 마지막 함수 반환값
      lastArgs = args; // 마지막 매개변수
    }
    return lastResult;
  }; as unknown as T;
}

// this를 추가한 경우
function cacheLastWithThis<T extends Function>(fn: T): T {
  let lastArgs: any[] | null = null;
  let lastResult: any;
  let lastContext: any; // lastContext 변수를 추가하여 마지막 호출 시의 this 값을 저장

  return function (this: any, ...args: any[]) {
    if (!lastArgs || lastContext !== this || !shallowEqual(lastArgs, args)) {
      lastResult = fn.apply(this, args); // 마지막 함수 반환값
      lastArgs = args; // 마지막 매개변수
      lastContext = this; // 마지막 this 값
    }
    return lastResult;
  } as unknown as T;
}
```

예시B: shallowObjectEqual

객체가 같은 지 체크하기 위한 for in 반복문의 **(b as any)[key]** 단언은 안전합니다
조건문에서 앞서 **key in b** (키가 진짜 객체 b에 있는지) 를 먼저 확인했기 때문입니다

```
function shallowObjectEqual<T extends object>(a: T, b: T): boolean {
  // a, b: 비교할 객체
  for (const [key, value] of Object.entries(a)) {
    // 객체 `a`의 모든 키 값 쌍을 순회
    if (!(key in b) || value !== (b as any)[key]) {
      // 객체 b에, 현재 키가 존재하는 지 확인.
      // 키가 없거나, 객체 a의 값 value과 다르다면? false를 반환한다.
      return false;
    }
  }
  return Object.keys(a).length === Object.keys(b).length;
  // 두 객체의 키 갯수가 같은 지 체크
  // 모든 키와 값이 일치하고, 키 개수가 동일하다 -> true (얕은 비교가 참)
}
```

Hide Unsafe Type Assertions in Well-Typed Functions

때때로 불가피하거나 편의를 위해
any를 사용해야 하는 경우가 있습니다
이 때 **any**를 올바른 시그니처의 함수 안에 숨겨서
함수 바깥의 타입 안정성 유지할 수 있습니다

타입 에러를 수정하기 위해
함수의 타입 시그니처를 바꾸지 않고,
유지하면서 타입 오류를 해결해야 합니다

41.

any의 진화란?

Understand Evolving Types

any는
진화합니다.

일반적인 변수의 타입은? 변수를 선언할 때 결정됩니다
any는 진화합니다

```ts
function range(start: number, limit: number) {
  const nums = [];
  //    ^? const nums: any[]
  for (let i = start; i < limit; i++) {
    nums.push(i);
    // ^? const nums: any[]
  }
  return nums;
  //     ^? const nums: number[]
  // number 값을 넣는 순간 진화한다
}
```

타입이 변화한다는 점에서 타입 좁히기 개념과 혼동할 수 있습니다
type narrowing와 any evolving의 다른 점은?
다양한 타입의 요소를 넣으면 타입이 확장되며 진화한다는 점입니다

```ts
const result = [];
//    ^? const result: any[]
result.push("a");
result;
// ^? const result: string[]
result.push(1);
result;
// ^? const result: (string | number)[]
```

진화 예시: 초기값이 null인 경우

try/catch 블록 안에서 변수를 할당하는 경우

```
let value = null;
//  ^? let value: any
try {
  value = doSomethingRiskyAndReturnANumber();
  value;
  // ^? let value: number
} catch (e) {
  console.warn("alas!");
}
value;
// ^? let value: number | null
// 진화
```

if else문의 경우

```
let value;
//  ^? let value: any
if (Math.random() < 0.5) {
  value = /hello/;
  value;
  // ^? let value: RegExp
} else {
  value = 12;
  value;
  // ^? let value: number
}
value;
// ^? let value: number | RegExp
// 진화함
```

특징

any 타입 진화는 암시적 **any**인 값에 적용되며, 명시적 **any**는 진화 없이 유지됩니다

값을 할당하거나, 배열에 요소를 넣은 후에만 일어나기 때문에 편집기에서 이상하게 보일 수 있습니다
할당이 일어난 줄의 타입을 조사해도 any입니다. 할당이 끝난 후 진화하기 때문입니다

어떠한 할당도 하지 않고 사용하면 **noImplicitAny** 오류가 발생합니다
암시적 any 상태일 때 값을 읽으려 하면 오류가 발생합니다

예시: 함수 호출을 거쳤을 때

함수 호출을 거쳐도 진화하지 않습니다

```
function makeSquares(start: number, limit: number) {
  const nums = [];
  //    ~~~~ Variable 'nums' implicitly has type 'any[]' in some locations
  range(start, limit).forEach((i) => {
    nums.push(i * i); // 진화하지 않는다
  });
  return nums;
  //     ~~~~ Variable 'nums' implicitly has an 'any[]' type
}
```

예시에서 **.forEach** 함수 호출을 거쳤을 때는 **any**가 진화하지 않습니다
forEach로 순회하는 대신, 배열의 **map**과 **filter** 메서드로 단일 구문으로 배열 생성해서
any 전체를 진화시키는 방법을 생각할 수 있습니다

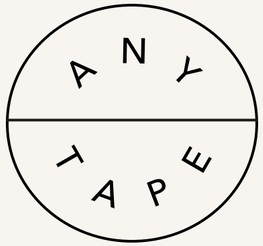
진화가 되는 방식은, 일반적인 변수가 추론되는 원리와 동일합니다

만약 진화한 배열의 타입이 **(string|number)[]**이라면
number[]이어야 하는 데 **string**이 섞였여서 진화했을 가능성도 있습니다
따라서 명시적 타입 구문이 더 안전성 있는 좋은 설계입니다

Understand Evolving Types

일반적으로 타입은 정제 **refine**만 되지만
암시적 **Any**인 경우 **(null, undefined, []**로 초기화된 경우)
타입은 진화합니다

진화가 일어남을 알고 있다면
필요한 경우 타입 구문을 줄일 수 있다.
다만 에러 체킹과 타입 안정성을 위해서는
타입을 진화 시키는 것보다 명시적인 타입 구문을 사용하는 것이 낫습니다



Thank you.

이펙티브 타입스크립트, Dan Vanderkam - 5장, any 다루기

예제 코드: https://github.com/danvk/effective-typescript