TYPE INFERENCE

타입 추론

2024.05.26 두선아

TABLE OF CONTENTS

ITEM 19

Avoid Cluttering Your Code with Inferable Types

추론 가능 타입을 사용해 장황한 코드 방지하기

타입 추론

명시적 타입 구문 없이도
타입을 추론할 수 있습니다

모든 변수에 타입 선언하는 것은 비생산적, 불필요 합니다

타입 추론이 된다면 명시적 타입 구문은 필요하지 않습니다

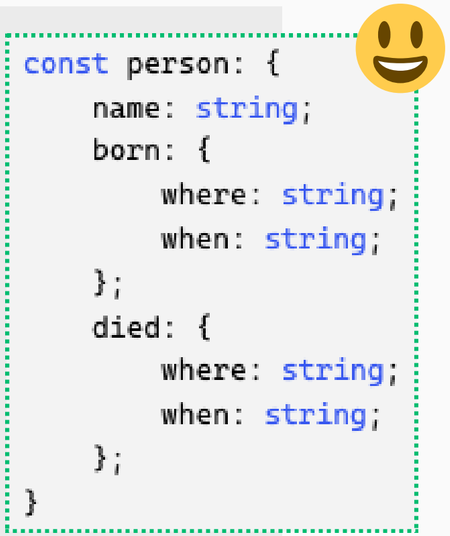
```
// not good!
let x: number = 12;

// enough!
let x = 12;
```

복잡한 객체 또는 함수의 반환 타입도 추론할 수 있습니다

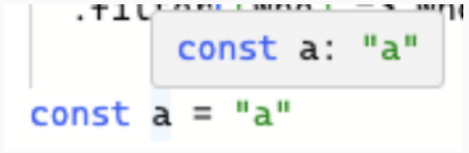
```
// 객체 추론
const person = {
  name: "Sojourner Truth",
  born: {
    where: "Swartekill, NY",
    when: "c.1797",
  },
  died: {
    where: "Battle Creek, MI",
    when: "Nov. 26, 1883",
  },
};

// 함수의 반환 타입 추론
function square(nums: number[]) {
  return nums.map((x) => x * x);
}
const squares = square([1, 2, 3, 4]);
//    ^? const squares: number[]
```



타입 추론 예시

유닛 타입



const로 상수를 만들어 유닛 타입을 추론할 수 있습니다.
보다 정확한 타입 추론을 통해 타입 오류 방지!

함수 타입

```
function logProduct(product: Product) {
```

함수 매개변수에 타입 정보가 필요한 이유

타입스크립트의 타입은 처음 등장할 때 결정됩니다.
최종 사용처까지 고려하지 않기 때문에 명시적 타입 선언이 필요합니다

구조분해 할당 (비구조화 할당)

지역 변수의 타입이 추론되도록 할 수 있습니다.
명시적 타입 선언을 하지 않아도 됩니다!

```
function logProduct(product: Product) {
  const id: number = product.id;
  // ~~ Type 'string' is not assignable to type 'number'
  const name: string = product.name;
  const price: number = product.price;
  console.log(id, name, price);
}
```

구조분해 할당

```
function logProduct(product: Product) {
  const {id, name, price} = product
  console.log(id, name, price);
}
```

매개변수 기본값이 있는 경우, 타입을 추론한다

```
function parseNumber(str: string, base = 10) {
  //                              ^? (parameter) base: number
  // ...
}
```

타입 정보가 있는 라이브러리

타입 지원하는 라이브러리에서 콜백 함수의 매개변수 타입은 보통 자동으로 추론됩니다

```
// Don't do this:
app.get("/health", (request: express.Request, response: express.Response)
=> {
  response.send("OK");
});

// Do this:
app.get("/health", (request, response) => {
  //                 ^? (parameter) request: Request<...>
  response.send("OK");
  // ^? (parameter) response: Response<...>
});
```

Request, Response

lodash DT

4.17.21 • Public • Published 3 years ago

```
interface FoodItem {
  icon: string;
  category: string;
  price: number;
}

const foodItems: FoodItem[] = [
  { icon: "🍔", category: "Fast Food", price: 5000 },
  { icon: "🍕", category: "Fast Food", price: 8000 },
  { icon: "🍣", category: "Japanese", price: 12000 },
  { icon: "🍜", category: "Japanese", price: 10000 },
  { icon: "🍝", category: "Italian", price: 15000 },
  { icon: "🥗", category: "Healthy", price: 9000 },
];

// Lodash의 groupBy를 사용할 때 콜백 함수의 매개변수 타입이 자동으로 추론됨
const groupedByCategory = _.groupBy(foodItems, (item) => item.category);
```

(FoodItem[], (item: FoodItem) => FoodItem) => Record<string, FoodItem[]>

npm i

타입을
명시하고
싶은 경우

객체 리터럴 정의

정의에 타입을 명시하면 excess property check (추가 속성 체크, 잉여 속성 체크)가 동작합니다

- 선택적 속성이 있는 타입의 오타 등의 오류
- 변수가 사용되는 순간이 아닌 할당하는 시점에 오류를 표시

함수 반환값

타입 추론이 가능할지라도, 구현상의 오류를 함수 내에서 확인하기 위해 타입 구문을 명시할 수 있습니다.
(함수를 호출한 곳까지 영향을 미치지 않기 위해서입니다!)

```ts
const cache: { [ticker: string]: number } = {};
function getQuote(ticker: string) {
  if (ticker in cache) {
    return cache[ticker]; // 구현 오류: Promise를 반환해야 한다!
  }
  return fetch(`https://quotes.example.com/?q=${ticker}`)
    .then((response) => response.json())
    .then((quote) => {
      cache[ticker] = quote;
      return quote as number;
    });
}

// 함수를 호출한 곳에서 에러가 발생한다
getQuote("MSFT").then(considerBuying);
//               ~~~~ Property 'then' does not exist on type
//                    'number | Promise<number>'
```

```ts
// 의도된 반환 타입을 명시하여 => 에러가 발생한 위치를 정확히 표기하기
const cache: { [ticker: string]: number } = {};
function getQuote(ticker: string): Promise<number> {
  if (ticker in cache) {
    return cache[ticker];
    // ~~~ Type 'number' is not assignable to type 'Promise<number>'
  }
  // ...
}
```

- **오류의 위치를 제대로 표시한다**
- **함수 시그니처를 더욱 명확하게 한다 (입/출력 타입 명시)**
- **명명된 타입 사용하기**

ITEM 20

Use Different Variables for Different Types

다른 타입에는 다른 변수 사용하기

다른 타입에는 별도의 변수를 사용하는 게 바람직

타입스크립트에서 변수의 값은 바뀔 수 있지만 그 타입은 보통 바뀌지 않습니다

멈춰!✋

```ts
let productId: string | number = "12-34-56";
fetchProduct(productId);

// 타입을 좁혀서 다른 타입을 사용할 수 있습니다.
productId = 123456; // OK
fetchProductBySerialNumber(productId); // OK

// 다른 변수를 사용하는 것이 좋습니다
const productId = "12-34-56";
fetchProduct(productId);

const serial = 123456; // OK
fetchProductBySerialNumber(serial); // OK

// 스코프
const productId = "12-34-56";
fetchProduct(productId);

{
  const productId = 123456; // OK
  fetchProductBySerialNumber(productId); // OK
}
```

shadowed 변수

장점

- 서로 관련이 없는 두 개의 값을 분리할 수 있습니다.
- 변수명을 더 구체적으로 지을 수 있습니다
- 타입 추론을 향상시키며, 불필요한 타입 구문을 작성하지 않습니다.
- 타입이 간결해집니다.
- let 대신 const로 선언하여, 간결하고 타입을 추론하기 쉬운 코드를 작성합니다.

결론

- 타입이 바뀌는 변수를 피하자
- 목적이 다른 곳에는 별도의 변수명을 사용하자

코드를 명확하게 작성하자

ITEM 21

Understand How a Variable Gets Its Type

타입 넓히기

Type Widening

런타임에 모든 변수는 유일한 값을 가집니다.
정적 분석 시점(타입스크립트가 작성된 코드를 체크하는 시점)에
변수는 **가능한 값들의 집합**인 **타입**을 가집니다.

지정된 단일 값들을 가지고, 할당 가능한 값들의 집합을 유추합니다.
만약 정보가 충분하지 않으면, 어떤 타입으로 추론되어야하는 지 알 수 없습니다 (작성자의 의도를 추측합니다)

```ts
interface Vector3 {
  x: number;
  y: number;
  z: number;
}
function getComponent(vector: Vector3, axis: "x" | "y" | "z") {
  return vector[axis];
}

let x = "x"; // const로 선언할 시 "x" 유닛 타입
let vec = { x: 10, y: 20, z: 30 };
getComponent(vec, x) 현재 x에 할당 가능한 값들의 집합은 string입니다.
//                   ~ Argument of type 'string' is not assignable
//                     to parameter of type '"x" | "y" | "z"'
```

타입 추론?

Type widening을 제어할 수 있는 방법

타입을 **더 좁은 범위로 추론하도록 제어**하는 방법에는 여러 가지가 있습니다.
변수나 상수의 타입을 명시적으로 지정하거나, 타입 체커에 추가적인 정보를 제공하여 가능한 한 좁은 타입으로 추론하게 할 수 있습니다.

1. const로 변수를 선언하면 더 좁은 타입이 됩니다.

- const를 사용하면 변수는 재할당이 불가능해지므로 더 좁은 타입으로 추론됩니다.
- 그러나 객체와 배열의 경우, 내부 요소들은 여전히 넓은 타입으로 추론될 수 있습니다.
- 객체에 다른 속성을 추가할 수 없으므로 객체를 한 번에 만들어야 합니다.

2. 타입 체커에 추가적인 문맥을 제공합니다. (함수의 매개변수로 값 전달 등)

```
function processPoint(point: { x: 10; y: 20 }) {
  // processPoint 내부에서는 point의 타입이 { x: 10, y: 20 }로 추론됩니다.
}
const p = { x: 10, y: 20 };
processPoint(p);
```

3. const 단언문을 사용 (as const)

- 값 뒤에 as const를 작성하면, 타입스크립트는 최대한 좁은 타입으로 추론합니다.

```
const point = { x: 10, y: 20 } as const; // Type is { readonly x: 10, readonly y: 20 }
```

Type widening을 제어할 수 있는 방법

4.satisfies 키워드를 사용

```
type Point = [number, number];
const capitalsBad = {
  ny: [-73.7562, 42.6526, 148],
  //  ~~ Type '[number, number, number]' is not assignable to type 'Point'.
  ca: [-121.4944, 38.5816, 26],
  //  ~~ Type '[number, number, number]' is not assignable to type 'Point'.
} satisfies Record<string, Point>;
```

satisfies

satisfies 키워드는 capitalsBad 객체가 **Record<string, Point>** 타입을 만족하는지 확인

- 이 경우, 각 속성의 값이 Point 타입(즉, [number, number])을 충족하지 않으면 오류 발생

특징

- satisfies 키워드는 타입 확인을 수행하지만, 변수의 원래 타입을 변경하지 않음
- as와 달리, satisfies는 타입 단언을 하지 않으므로 타입 안전성을 유지

한글 서적에는 삭제되었지만 알아두자!

ITEM 22

Understand Type Narrowing

타입 좁히기

Type narrowing

타입을 좁히는 방법

null 체크하기

```
const elem = document.getElementById("what-time-is-it");
//    ^? const elem: HTMLElement | null
if (!elem) throw new Error("Unable to find #what-time-is-it"); // null을 체크함
elem.innerHTML = "Party Time".blink();
// ^? const elem: HTMLElement
```

attribute를 체크하여 interface 타입을 좁히기

```
// [속성 체크] 속성의 존재 여부로 interface 타입 좁히기
interface Apple {
  isGoodForBaking: boolean;
}
interface Orange {
  numSlices: number;
}
function pickFruit(fruit: Apple | Orange) {
  if ("isGoodForBaking" in fruit) {
    fruit;
    // ^? (parameter) fruit: Apple
  } else {
    fruit;
    // ^? (parameter) fruit: Orange
  }
  fruit;
  // ^? (parameter) fruit: Apple | Orange
}
```

값의 타입을 instacneof를 사용하여 확인하기

```
// [분기문] 값의 타입을 instacneof를 사용하여 좁힘
function contains(text: string, search: string | RegExp) {
  if (search instanceof RegExp) {
    return !!search.exec(text);
    //       ^? (parameter) search: RegExp
  }
  return text.includes(search);
  //                   ^? (parameter) search: string
}
```

메소드를 활용하여 타입 좁히기

```
// [내장 함수]
function contains(text: string, terms: string | string[]) {
  const termList = Array.isArray(terms) ? terms : [terms]; // 내장 메소드로 타입 좁히기
  //    ^? const termList: string[] 이제 string 배열
  // ...
}
```

Type narrowing

타입을 좁히는 방법

태그된 유니온으로 확인하기

```ts
// [tagged union] 태그된 유니온, 또는 구별된 유니온 discriminated union
interface UploadEvent {
  type: "upload"; // 유니온!
  filename: string;
  contents: string;
}
interface DownloadEvent {
  type: "download"; // 유니온!
  filename: string;
}
type AppEvent = UploadEvent | DownloadEvent;

function handleEvent(e: AppEvent) {
  switch (e.type) {
    case "download": // 활용
      console.log("Download", e.filename);
      //                       ^? (parameter) e: DownloadEvent
      break;
    case "upload":
      console.log("Upload", e.filename, e.contents.length, "bytes");
      //                     ^? (parameter) e: UploadEvent
      break;
  }
}
```

사용자 정의 타입 가드

```ts
// 사용자 정의 타입 가드
function isInputElement(el: Element): el is HTMLInputElement {
  return "value" in el;
}

function getElementContent(el: HTMLElement) {
  if (isInputElement(el)) {
    // 인풋 엘리먼트이냐?
    return el.value;
    //     ^? (parameter) el: HTMLInputElement
  }
  return el.textContent;
  //     ^? (parameter) el: HTMLElement
}
```

사용자 정의 타입 가드, 예시!

```ts
// 커스텀 타입 가드 예시
function isDefined<T>(x: T | undefined): x is T {
  return x !== undefined;
}

const jackson5 = ["Jackie", "Tito", "Jermaine", "Marlon", "Micheal"];
const members = ["Janet", "Micheal"]
  .map((who) => jackson5.find((n) => n === who))
  .filter(isDefined); // filter(x => x !== undefined)은 반환되는 타입에 영향을 주지 않음
```

isDefined()로 undefined를 거릅니다

코드 작성 예시!

Type narrowing

꼼꼼히 살피기

오류A: null은 'object'입니다!

```ts
const elem = document.getElementById("what-time-is-it");
//    ^? const elem: HTMLElement | null
if (typeof elem === "object") {
  elem;
  // ^? const elem: HTMLElement | null 🤷‍♀️
}
```

오류B: ""와 0은 강제 변환 시 false입니다.

```ts
function maybeLogX(x?: number | string | null) {
  if (!x) {
    console.log(x);
    //          ^? (parameter) x: string | number | null | undefined 🤷‍♀️
  }
}
```

기억하기!

분기문 외에도 코드의 제어 흐름을 살펴보며
타입스크립트가 타입을 좁히는 과정을 이해해야 합니다.
Type Narrowing을 돕기 위해 태그된 유니온과, 사용자 정의 타입 가드를 사용할 수 있습니다.

ITEM 23

Create Objects All at Once

한꺼번에 객체 생성하기

객체와 타입 추론

변수의 값은 변경 가능하지만
타입스크립트의 타입은 일반적으로 변경되지 않습니다.

객체를 생성할 때는 속성을 하나씩 추가하기 보다는
여러 속성을 포함하여 한꺼번에 생성해야 타입 추론에 유리합니다

객체의 생성과 타입 추론

```ts
const pt = {}; // 변수의 타입이 {} 기준으로 추론됩니다.
//    ^? const pt: {}
pt.x = 3;
// ~ Property 'x' does not exist on type '{}'
//   존재하지 않는 속성을 추가할 수 없습니다
pt.y = 4;
// ~ Property 'y' does not exist on type '{}'

// interface
interface Point {
  x: number;
  y: number;
}
const pt: Point = {}; // 속성이 없다!
// ~~ Type '{}' is missing the following properties from type 'Point': x, y
pt.x = 3;
pt.y = 4;
```

타입 추론하기!

```ts
// 해결A: 한 번에 객체 생성하기
const pt: Point = {
  x: 3,
  y: 4, // 한 번에 정의하기!
};

// 해결B: 단언
const pt = {} as Point;
//    ^? const pt: Point
pt.x = 3;
pt.y = 4; // OK
```

객체의 타입을 정의할 때

객체 전개 연산자 (spread) 사용하기

객체에 속성을 추가하고, 타입스크립트가 새로운 타입을 추론할 수 있게 하는 방법입니다.

새로운 타입을 추론

```
const pt0 = {};
const pt1 = { ...pt0, x: 3 };
const pt: Point = { ...pt1, y: 4 }; // OK
```

y 속성을 추가!

조건부 속성을 추가하기

```
declare let hasMiddle: boolean;
const firstLast = { first: "Harry", last: "Truman" };
const president = { ...firstLast, ...(hasMiddle ? { middle: "S" } : {}) };
```

{} 또는 null로 객체 전개를 사용한다!

전개 연산자로 여러 속성을 추가하기

```
function addOptional<T extends object, U extends object>(
  a: T,
  b: U | null
): T & Partial<U> {
  return { ...a, ...b };
}
```

기억하기!

속성을 제각각 추가하지 말고, 한 번에 객체로 만들어서 타입을 추론합니다.
여러 오브젝트는 스프레드 문법을 사용해서 안전한 타입으로 추가할 수 있습니다.
객체에 조건부 속성(optional attribute)을 추가하는 법을 알아둡시다.

CODE EXAMPLE

타입 개선 예시

타입 개선 예시

런타임에 유저 페이지에서 “userId is null”이라는 구문의 오류가 발생하여
확인하고 오류와 타입을 수정한 예시입니다.

AS IS

```ts
// 런타임에 userId is null 오류 발생!
export const loadUserProfile = (queryClient: QueryClient) => async () => {
  const userId = localStorage.getItem("userId");

  const preloadUserProfile = await queryClient.ensureQueryData(
    getUserProfile(userId || "")
  );
  // eslint-disable-next-line
  // return type 단언때문에 eslint disable 사용했습니다
  let groupProfile: AxiosResponse<any, any> | null = null;

  const groupId = preloadUserProfile.data.group;
  if (groupId) {
    groupProfile = await queryClient.ensureQueryData(
      getGroupProfile(preloadUserProfile.data.organizationId, groupId)
    );
  }

  return { preloadUserProfile, groupProfile };
};
```

! eslint-disable

! noInplicitAny

```ts
const getUserProfile = async (userId: string) => {
  if (!userId) return Promise.reject("userId is null");
  return await api.get(`users/${userId}`);
};
```

에러 발생 위치

TO BE

```ts
// 타입 개선
export const loadUserProfile = (queryClient: QueryClient) => async () => {
  const userId = localStorage.getItem("userId");
  if (!userId) {
    logout();
    throw new Error("userId is not found in localStorage");
  }

  const preloadUserProfile = await queryClient.ensureQueryData(
    getUserProfile(userId)
  );
  let preloadGroupProfile = null;

  const groupId = preloadUserProfile.data.group;
  if (groupId) {
    preloadGroupProfile = await queryClient.ensureQueryData(
      getGroupProfile(preloadUserProfile.data.organizationId, groupId)
    );
  }

  return { preloadUserProfile, preloadGroupProfile };
};
```

에러가 throw되면 /error 페이지로 이동하도록 ErrorBoundary 컴포넌트를 감싸서 사용하고 있기 때문에 에러를 던집니다🤔 (컴포넌트가 아니기 때문에 useNaviate 혹을 사용할 수 없습니다)

! 라이브러리의 반환 타입을 사용하여 타입 체커가 타입을 추론하도록 합니다

AxiosResponse

AxiosResponse | null

```ts
export default function MyPage() {
  const { preloadUserProfile, preloadGroupProfile } = useLoaderData();
  if (!preloadUserProfile.data.id) logoutWithError();

  const {
    data: { data: userProfile },
  } = useQuery({
    ...getUserProfile(preloadUserProfile.data.id || ""),
    initialData: preloadUserProfile,
  });
  if (!userProfile.data.id) logoutWithError();
}
```

컴포넌트 단에도 예외 처리 추가

THANK YOU!

reference

이펙티브 타입스크립트, Dan Vanderkam
  3장, 타입 추론
  예제 코드: https://github.com/danvk/effective-typescript