



12장 타입스크립트 프로젝트관리

12.1 앰비언트 타입 활용하기

1. 앰비언트 타입 선언

앰비언트 타입 선언

대표적인 앰비언트 타입 선언 활용 사례

JS로 작성된 라이브러리

TS로 작성된 라이브러리

JS 어딘가에 전역 변수가 정의되어 있음을 TS에 알릴 때

2. 앰비언트 타입 선언 시 주의점

TS로 만드는 라이브러리에는 불필요

전역으로 타입을 정의하여 사용할 때 주의점

3. 앰비언트 타입 선언을 잘못 사용 시 문제점

4. 앰비언트 타입 활용하기

타입을 정의하여 임포트 없이 전역으로 공유

declare type 활용하기

declare module 활용하기

declare namespace 활용하기

declare global 활용하기

5. declare와 번들러의 시너지

12.2 스크립트와 설정 파일 활용하기

1. 스크립트 활용하기

실시간으로 타입을 검사하자

타입 커버리지 확인하기

2. 설정 파일 활용하기

TS 컴파일 속도 높이기

3. 에디터 활용하기

에디터에서 TS 서버 재시작하기

12.3 TS 마이그레이션

12.4 모노레퍼

1. 분산된 구조의 문제점

2. 통합할 수 있는 요소 찾기

3. 공통 모듈화로 관리하기

4. 모노레포의 탄생

모노레포 관리 장점

모노레포 관리 단점

 우형 이야기 : 모노레포 느낀점 



12.1 앰비언트 타입 활용하기

1. 앰비언트 타입 선언

`.ts` `.tsx` `.d.ts` 확장자를 가진 파일에서 TS의 타입 선언을 할 수 있다.

앰비언트 타입 선언

앰비언트 : '주변의'라는 사전적인 의미

`.d.ts` 확장자를 가진 파일에서는 **타입 선언만** 할 수 있으며, 값을 표현할 수 없다.

- `.d.ts` : 값을 포함하는 일반적인 선언과 구별하기 위한 확장자
- 앰비언트 타입 선언** : `.d.ts` 확장자를 가진 파일에서 하는 타입 선언

앰비언트 타입 선언 : TS에게 **JS 코드 안에서 이러한 정보들이 있어** 라고 알려주는 도구

앰비언트 타입 선언으로 값을 정의할 수는 없지만, `declare` 키워드를 사용하여 JS 값이 존재한다는 사실을 선언할 수 있다.

`declare` : TS 컴파일러에 어떤 것의 존재 여부를 명시해주는 역할 **'이러한 것이 존재해'** . 단순히 존재 여부만 알려주기 때문에 컴파일 대상이 아니다.

💡 TS 컴파일러에 타입 정보를 알려주는 `declare`

대표적인 앰비언트 타입 선언 활용 사례

TS를 사용하다보면 `.js` `.ts` 형식이 아닌 파일을 임포트할 때 종종 에러가 발생한다.

ex) JS로 png 등 이미지 파일을 모듈로 임포트할 때, TS 환경에서는 에러가 발생한다.

fdvdfvdsfv

❗ TS는 기본적으로 `.ts` `.js` 파일만 이해하고 그 외의 다른 파일 형식은 인식하지 못한다. 그래서 알지 못하는 파일 형식을 모듈로 가져오려 하면 에러가 발생한다.


👤 TS의 `declare` 키워드를 사용하여 특정 형식을 모듈로 선언하면 TS 컴파일러에 미리 정보를 제공한다.

```
declare module "*.png" {
  const src: string;
  export default src;
}
```

JS로 작성된 라이브러리

TS에서 JS로 작성된 라이브러리를 사용하기 위해서는 타입 선언이 없으므로 임포트한 모듈이 모두 any로 추론된다.

! tsconfig.json 파일에서 any를 사용 못하게 설정 → 프로젝트 빌드 X

 **앰비언트 타입 선언**

JS 라이브러리 내부 함수와 변수의 타입을 앰비언트 타입으로 선언하면, TS는 자동으로 `.d.ts` 확장자를 가진 파일을 검색하여 타입 검사를 진행하기 때문에 문제없이 컴파일된다.

- VSCode에서도 `.d.ts` 확장자 파일을 해석하여 코드 작성 시, 유용한 타입 힌트 제공


ex) @types/react를 npm install -D 명령을 통해 설치

→ node_modules/@types/react에 `index.d.ts` `global.d.ts`가 설치된다.

TS로 작성된 라이브러리

TS로 작성된 라이브러리일지라도 `JS 파일` 과 `.d.ts` 파일로 배포되는 것이 일반적이다.

- TS 파일을 직접 배포 시, 라이브러리 사용자가 TS를 컴파일 할 때 라이브러리 코드도 함께 컴파일한다.
- JS 파일과 .d.ts 파일로 배포 시, 라이브러리 코드를 따로 컴파일하지 않아도 된다. → 컴파일 시간이 크게 줄어든다.

 tsconfig.json 파일의 declaration을 true로 설정 시, TS 컴파일러는 자동으로 .d.ts 파일을 생성한다.

JS 어딘가에 전역 변수가 정의되어 있음을 TS에 알릴 때

ex) 웹뷰 개발 시, 네이티브 앱과의 통신을 위한 인터페이스를 네이티브 앱이 window 객체에 추가하는 경우가 많다.

전역 객체 `window` 에 변수나 함수를 추가하면 TS에 직접 구현하지 않았더라도 실제 런타임 환경에서 해당 변수를 사용할 수 있다.

네이티브 앱에서 Window

! Window 객체의 속성은 TS로 직접 정의한 값이 아니기 때문에, TS는 해당 속성이 Window 객체의 타입에 존재하지 X한다고 판단한다. 그래서 해당 속성에 접근 시, 에러 발생

👤 global namespace에 있는 Window 객체에 해당 속성이 정의되어 있다는 것을 나타내기 위해 `ambient 타입 선언` 사용

```
declare global {  
  interface Window {  
    deviceId: string | undefined;  
    appVersion: string;  
  }  
}
```

2. 앰비언트 타입 선언 시 주의점

TS로 만드는 라이브러리에는 불필요

💡 `tsconfig.json` 의 `declaration` 을 true로 설정 시, TS 컴파일러가 `.d.ts` 파일을 자동으로 생성해준다.

전역으로 타입을 정의하여 사용할 때 주의점

서로 다른 라이브러리에서 동일한 이름의 앰비언트 타입 선언 시 충돌 발생

3. 앰비언트 타입 선언을 잘못 사용 시 문제점

.ts 파일 내의 앰비언트 변수 선언 시, 앰비언트 타입의 의존성 관계가 보이지 않기 때문에 변경에 의한 영향 범위를 파악하기 어렵다. → 개발자에게 혼란 야기

- .ts, .tsx 파일 내에서의 앰비언트 타입 `declare` 선언

```
// src/index.tsx

import React from "react";
import ReactDOM from "react-dom";
import App from "App";

declare global {
  interface Window {
    Example: string;
  }
}


const SomeComponent = () => {
  return <div>앰비언트 타입 선언은 .tsx 파일에서도 가능</div>;
};
```

다른 파일에서도 임포트 없이 사용가능하다.

```
// src/test.tsx

window.Example; // 앰비언트 타입 선언으로 인해 타입스크립트 에러가 발생
```

! 앰비언트 변수 선언은 어느 곳에도 영향을 준다. → 일반 타입 선언과 섞이게 되면 앰비언트 선언이 어떤 파일에 포함되어 있는지 파악이 어려워진다.

 .d.ts 확장자 파일 내에서 앰비언트 타입 선언을 하는 것은 일종의 개발자 간의 약속이다.

4. 앰비언트 타입 활용하기

타입을 정의하여 임포트 없이 전역으로 공유

.d.ts 파일에서의 앰비언트 타입 선언 = 전역 변수

유용한 유틸리티 타입을 앰비언트 타입으로 선언하면 모든 코드에서 임포트하지 않아도 해당 타입을 사용할 수 있다. **내장 타입 유틸리티 함수**

```
// src/index.d.ts
type Optional<T extends object, K extends keyof T = keyof T> =
  Partial<Pick<T, K>>;
  // T - 객체 타입만 받도록 제한
  // K - 기본값으로 모든 키가 지정됨
  // Omit<T, K> - T에서 K로 지정된 프로퍼티들을 제외한 나머지
  // Partial<Pick<T, K>> - K로 지정된 프로퍼티들만 선택하여 옵셔널로
  // & : 두 타입 합치기

// src/components.ts
type Props = { name: string; age: number; visible: boolean };
type OptionalProps = Optional<Props>; // Expect: { name?: str

type PartialNameProps = Optional<Props, 'name'>; // 결과: { na
```



▼ 왜 여기에는 `declare` 키워드를 사용하지 않은걸까?

- `declare` 키워드가 필요한 경우
변수, 함수, 클래스 등의 값을 선언할 때
- `declare` 키워드가 필요하지 X은 경우
타입과 인터페이스 선언



`.d.ts` 파일에 위치하는 것만으로도 전역 타입으로 선언된다.

declare type 활용하기

보편적으로 많이 사용하는 커스텀 유틸리티 타입을 `declare type` 으로 선언하여 전역에서 사용한다.

ex) Nullable 타입 선언

```
declare type Nullable<T> = T | null;

const name: Nullable<string> = "woowa";
```

declare module 활용하기

CSS-in-JS 라이브러리의 사례

theme의 인터페이스 타입을 확장 → theme 타입이 자동으로 완성되도록 하는 기능이 있다.

기존의 폰트, 크기, 색상 등을 객체로 관리한다.

이렇게 정의된 theme에서 스타일 값을 가져와 기존 인터페이스 타입과 통합하여 theme 타입이 자동으로 완성되는 기능을 지원한다.

```
// src/styles/theme.ts
// 1. 테마 객체 정의
const fontSizes = {
  xl: "30px",
  // ...
};

const colors = {
  gray_100: "#222222",
  gray_200: "#444444",
  // ...
};

const depths = {
  origin: 0,
  foreground: 10,
  dialog: 100,
  // ...
};

export const theme = {
  fontSizes,
  colors,
  depths,
};
```

```
// src/styles/styled.d.ts
// 2. styled-components의 DefaultTheme 인터페이스를 확장
declare module "styled-components" {
  type Theme = typeof theme;
  export type DefaultTheme = Theme; // 덮어쓰기
}
```

로컬 이미지나 SVG 같이 외부로 노출되어 있지 않은 파일을 모듈로 인식하여 사용

```
declare module "*.gif" {
  const src: string;
  export default src;
}
```



▼ declare vs. declare module

declare : 개별 변수, 함수, 클래스 등을 전역으로 선언

declare module : 모듈 자체를 선언하거나 or 기존 모듈의 타입을 확장

1. 새로운 모듈 선언
ex) gif 예시
2. 모듈 확장
ex) styled-component 예시

declare namespace 활용하기

Node.js 환경에서 `.env` 파일을 사용할 때, `declare namespace` 를 활용하여 `process.env`로 설정값을 손쉽게 불러오고, 환경변수의 자동 완성 기능을 쓸 수 있다.

→ `as` 단언을 사용하지 않아도 된다.

```
declare namespace NodeJS {
  interface ProcessEnv {
```



```

    readonly API_URL: string;
    readonly API_INTERNAL_URL: string;
    // ...
  }
}

```

1. namespace를 활용하여 process.env 타입을 보강해주지 X은 경우

```

function log(str: string) {
  console.log(str);
}

// .env
API_URL = "localhost:8080";

log(process.env.API_URL as string);

```

2. namespace를 활용하여 process.env 타입을 보강한 경우

```

function log(str: string) {
  console.log(str);
}

// .env
API_URL = "localhost:8080";

declare namespace NodeJS {
  interface ProcessEnv {
    readonly API_URL: string;
  }
}

log(process.env.API_URL);

```

declare global 활용하기

`declare global` : 전역 변수 선언 시 사용

ex) 전역 변수인 Window 객체의 스코프에서 사용되는 모듈이나 변수 추가

- 네이티브 앱과의 통신을 위한 인터페이스를 Window 객체에 추가할 때 활용 가능

```
declare global {  
  interface Window {  
    newProperty: string; // Window 객체에 newProperty 속성을 추가  
  }  
}
```

5. declare와 번들러의 시너지

`declare global` 로 전역 변수 선언하는 과정 + 번들러를 통해 데이터를 주입하는 절차를 함께 활용하면 시너지가 난다.

전역에 `_color` 변수가 존재함을 TS 컴파일러에 알리면 해당 객체를 활용할 수 있다.

```
const color = {  
  white: "#ffffff",  
  black: "#000000",  
} as const;  
  
type ColorSet = typeof color;  
  
// 전역 선언  
declare global { // 타입만 존재하고 실제 값은 없음  
  const _color: ColorSet;  
}  
  
// 사용  
const white = _color["white"]; // _color는 실제로 존재 x
```

! ColorSet 타입을 가지고 있는 `_color` 객체의 실제 데이터는 아직 존재하지 않는다.



번들 시점 에 번들러 를 통해서 해당 데이터를 주입한다.

번들러 : 웹 애플리케이션을 구성하는 수많은 파일(JS, CSS, 이미지..) 들을 하나 또는 여러 개의 파일로 합쳐주는 도구

ex) Webpack, Vite, Rollup ..

번들 시점 : 개발한 코드가 실제 배포용 파일로 변환되는 시점. 즉, 번들러가 파일들을 합치고 처리하는 시점

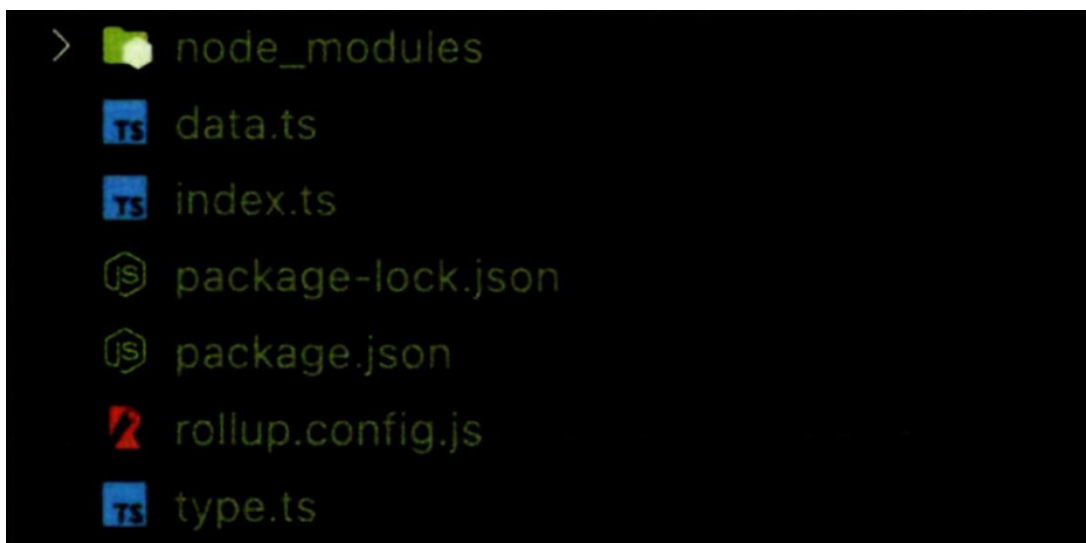
- 롤업 번들러의 `inject` 모듈로 데이터를 주입하는 예시

inject 모듈

inject는 임포트문의 경로를 분석하여 데이터를 가져온다.

```
import {color} from "./data";
```

./data 경로에서 color를 가져오는 경우, ['./data', 'color']로 지정하여 어떤 데이터 값을 가져올지 명시할 수 있다.



전체 폴더 구조

롤업 번들러 설정 `rollup.config.js` 에서 `inject` 모듈을 사용하여 `_color`에 해당하는 데이터를 삽입한다.

```
// data.ts - 색상 정의
export const color = {
  white: "#ffffff",
```

```

    black: "#000000",
  } as const;

// type.ts - 해당 데이터로부터 타입 정의하여 전역 선언
import { color } from "./data";
type ColorSet = typeof color;
declare global {
  const _color: ColorSet;
}

// index.ts - 전역 타입으로 선언된 변수를 콘솔 출력
// _color["white"]가 color["white"] 처럼 동작!
console.log(_color["white"]);

// rollup.config.js - 롤업 번들러 설정
import inject from "@rollup/plugin-inject";
import typescript from "@rollup/plugin-typescript";
export default [
  {
    input: "index.ts",
    output: [
      {
        dir: "lib",
        format: "esm",
      },
    ],
    // _color를 실제 data.ts의 color 객체로 대체
    plugins: [typescript(), inject({ _color: ["./data", "colo
  },
];

```

```

→ lib git:(master) ✗ node lib/index.js
#ffffff

```



#ffffff 정상 출력

? 굳이 inject모듈을 사용하지 않고, declare global에 타입을 선언하고 타입에 맞게 객체를 미리 주입하면 안될까?

```
// 타입 선언과 동시에 실제 객체도 할당
declare global {
  const _color: {
    white: string;
    black: string;
  } = {
    white: "#ffffff",
    black: "#000000"
  };
}
```

X : 이 방식은 동작하지 않는다.

`declare` 는 타입 시스템에만 존재하는 선언을 할 때 사용하므로 실제 값을 할당하는 것은 할 수 없다.

-  타입만 선언
-  값 할당

12.2 스크립트와 설정 파일 활용하기

TS 프로젝트에서 스크립트와 tsconfig 등을 잘 활용하면 개발 생산성을 높일 수 있다.

TS 프로젝트 관리 시 유용한 꿀팁 대방출~!


1. 스크립트 활용하기

실시간으로 타입을 검사하자

에디터 `VSCode` 가 가능한 한 빠르게 타입 에러를 감지해준다.

! 컴퓨터 성능이 떨어지거나 프로젝트 규모가 커지면 타입 에러를 알려주는 속도가 느려진다.

- 검사하려는 특정 파일을 열어여만 타입 에러가 나타난다.
- 에디터에서 에러가 없었는데, 커밋 후 뒤늦게 깃훅 도구 **husky** 에 의해 타입 에러가 발견 된다.

 스크립트를 통해 실시간 에러 확인 가능

```
// 프로젝트의 tsc(타입스크립트 컴파일러)를 실행한다.
// 파일이 변경될 때마다 tsc가 실행되어 어디에서 타입 에러 발생했는지 실시간으로
yarn tsc --noEmit --incremental --watch
```

옵션 설명

- noEmit : JS로 된 출력 파일 생성 X도록 설정
- incremental : **증분 컴파일** 활성화하여 컴파일 시간 단축
- w : 파일 변경 사항 모니터링

증분 컴파일 : 매번 모든 대상을 컴파일하는 게 아니라 변경 사항이 있는 부분만을 컴파일한다. 활용 시, 컴파일 시간 ⬇️

```
[오후 9:21:31] Starting compilation in watch mode...
[오후 9:21:37] Found 0 errors. Watching for file changes.
```

타입 에러 X일 때

```
[오후 9:23:17] File change detected. Starting incremental compilation...
app/views/registration/DriverLicenseFormScreen.tsx:125:11 - error TS2322: Type '{ visible: boolean; onPressClose: () => void; wrongProps: number; }' is not assignable to type 'IntrinsicAttributes & Props'.
  Property 'wrongProps' does not exist on type 'IntrinsicAttributes & Props'.
125     wrongProps={1}
      ~~~~~
[오후 9:23:17] Found 1 error. Watching for file changes.
```

타입 에러 발생할 때

타입 커버리지 확인하기

현재 프로젝트에서 얼마나 TS를 적절하게 쓰고 있는지 확인한다.

프로젝트의 모든 부분이 TS 통제 하에 돌아가고 있는지를 정량적으로 판단한다.

```
// 타입 커버리지와 any를 사용하고 있는 변수의 위치가 나타난다.
npx type-coverage --detail
```



```
}
}
```

// 방법2. 스크립트 활용

```
yarn tsc --noEmit --incremental --diagnostic
```

• incremental 미적용 시

```
Files:           1027
Lines:           290570
Nodes:           942690
Identifiers:      288806
Symbols:          603014
Types:            134746
Instantiations:   406542
Memory used:      540032K
I/O read:         0.75s
I/O write:        0.00s
Parse time:       6.67s
Bind time:        1.59s
Check time:       15.99s
Emit time:        0.00s
Total time:       24.25s
```

• incremental 적용 시

```
Files:           1027
Lines:           290570
Nodes:           942690
Identifiers:      288806
Symbols:          202654
Types:            13161
Instantiations:   29010
Memory used:      321981K
I/O read:         0.62s
I/O write:        0.00s
Parse time:       6.43s
Bind time:        1.73s
Check time:       2.01s
Emit time:        0.18s
Total time:       10.36s
```


전체 시간 **Total time** : 기존보다 10초 이상 빨라졌다.
검사 시간

Check time : 특히 차이가 확연하다.

3. 에디터 활용하기

에디터에서 TS 서버 재시작하기

! JS IDE **VSCode** 에서 프로그래밍하다 보면, 정의된 타입이 있는 객체인데도 임포트되지 않거나, 자동 완성 기능이 동작 X는 경우가 있다.

 TS 서버 재실행

 **Restart TS server** 기능 : **cmd + shift + P**

12.3 TS 마이그레이션

TS 프로젝트를 새로 구축한 사례 > 기존 JS 프로젝트를 TS로 마이그레이션한 사례
→ 새로운 설계를 바탕으로 타입을 작성하는 게 더 효율적이기 때문이다.

12.4 모노레퍼

여러 프로젝트 관리 시, 일반적으로 개별 프로젝트마다 별도의 레포지토리를 생성하여 관리한다.

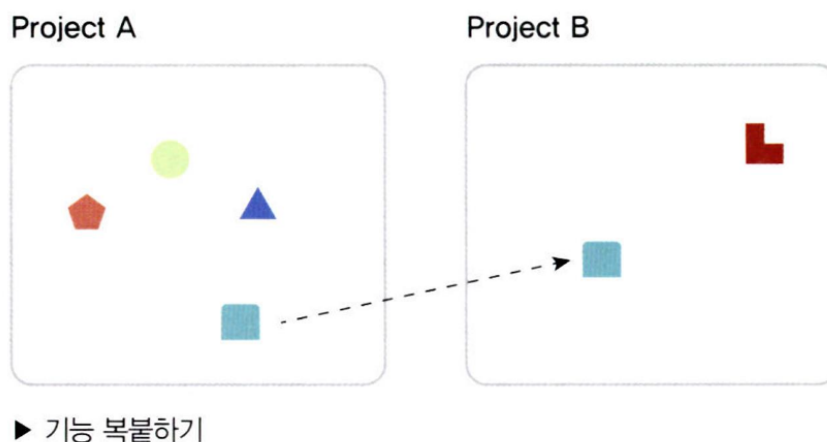
→ 공통적인 요소를 통합하면, 조금 더 효율적으로 관리할 수 있다.

1. 분산된 구조의 문제점

3개의 독립적인 프로젝트가 있다.

- 개발자는 각각의 레포지토리에서 해당 프로젝트를 위한 Jest, 바벨, ESLint, TS 등의 설정 파일을 별도 구성한다.
- 파이프라인, 공통 컴포넌트, 소스코드를 독립적으로 관리한다.

프로젝트에 필요한 기능이 다른 프로젝트에 존재하는 상황이다.



! 해당 기능을 복불하면 개발 시간을 아낄 수 있지만 프로젝트 관리 측면에서 어려움이 생긴다. → 개발자 경험 DX 저하

- 뒤늦게 새로운 버그 발견 or 기능 확장 위한 기능 수정 시, 프로젝트의 개수만큼 반복적인 수정 작업 필요

- 특정 라이브러리에 문제 생기거나 더이상 사용되지 X을 시, 모든 프로젝트에서 일일이 대응 필요

결과적으로, 분산된 구조는 생산성을 떨어뜨린다.

👤 반복되는 코드를 함수화하여 통합하듯이 한 곳에서 프로젝트를 관리할 수 있도록 통합한다.

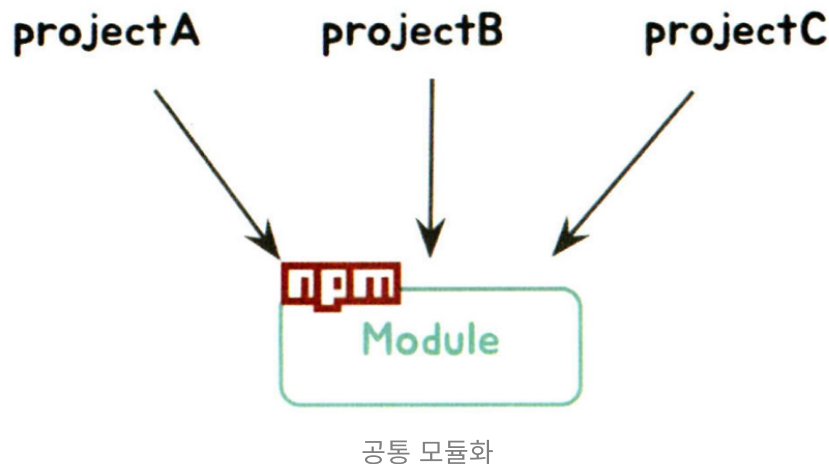
2. 통합할 수 있는 요소 찾기

이때 당연히 각 파일의 소스코드가 같지 않다면 통합을 위해 일부 수정해야 한다.

3. 공통 모듈화로 관리하기

패키지 관리자 `npm` 를 활용하여 공통 모듈을 생성하고 관리한다. → 각 프로젝트에서 간편하게 모듈과 의존성을 맺고 사용할 수 있게 된다.

- 새로운 프로젝트를 시작하더라도 모듈을 통해 재사용 가능
- 특정 기능의 변경이 필요할 때, 해당 모듈의 소스코드만 수정 가능 → 유지보수 👍



- ! 공통 모듈에 변경 발생 시, 해당 모듈을 사용하는 프로젝트에서도 추가 작업이 필요하다.
- ! 공통 모듈의 개수가 늘어나면, 관리할 레포지토리도 그만큼 늘어난다.
- ! 새로운 공통 모듈 필요 시, 개발자는 새로운 레포지토리를 생성하고, 개발 환경을 설정하며, 패키지 관리자를 사용하여 모듈을 게시해야 한다.
- ! 새로운 프로젝트를 시작할 때도 빌드를 위한 CI/CD 파이프라인, Lint, 테스트 등도 별도로 설정해야 한다.


4. 모노레포의 탄생

모노레포 : 버전 관리 시스템에서 여러 프로젝트를 하나의 레포지토리로 통합하여 관리하는 SW 개발 전략

이전에는 모놀리식 기법을 주로 사용했다.

모놀리식 : 다양한 기능을 가진 프로젝트를 하나의 레포지토리로 관리하는 기법

! 모놀리식 구조는 코드 간의 직접적인 의존이 발생하여 일부 로직만 변경될 때도 전체 프로젝트에 영향을 줄 수 있다.

 폴리레포 방식과 모노레포 방식 등장

폴리레포 : 거대한 프로젝트를 작은 프로젝트의 집합으로 나누어 관리하는 방식

모노레포 : 하나의 레포지토리로 모든 것을 관리하는 방식 (채택)

- 개발 환경 설정 통합 가능

모노레포 관리 장점

- 개발 환경 설정 **Lint** **CI/CD** 통한 관리 → 불필요한 코드 중복 X
- 공통 모듈도 동일한 프로젝트 내에서 관리 → 별도의 패키지 관리자를 통해 모듈 게시할 필요 X : 기능 변화 쉽게 추적, 의존성 관리

모노레포 관리 단점

- 시간이 지나면서 레포지토리가 거대해진다.
- 하나의 레포지토리에 여러 팀의 이해관계가 얹혀있다면 소유권, 권한 관리가 복잡해진다.

우형 이야기 : 모노레포 느낀점

- 모노레포 안에 있는 패키지의 개별적인 버저닝 필요
패키지들끼리 연관성이 높아서 한 곳에서 효율적으로 관리할 필요가 있다.
- 패키지 간의 연관관계가 있을 때 버저닝이 확실하게 보장된다.
 - 공통적인 부분을 새로운 패키지로 분리해야 할 때도 빠르게 작업 가능하다.
- 모든 패키지를 보게 되어 퍼포먼스가 떨어진다.
초창기 설정 시 공통 의존성이 꼬이는 문제가 발생한다. 오히려 더 많은 리소스 투입.

- 공통적인 코드 관리 측면에서 유용하다.
- 여러 부서에서 관리하다 보니 깃 로그를 읽고 변경 내역을 빠르게 파악하는 게 어렵다.
 - 모노레포로 합쳐져 있지만 배포 주기가 다르다.
- `pnpm`, `Lerna` + `npm workspace` 를 사용하여 모노레포를 관리한다.
- `yarn berry` 로 모노레포를 관리한다.
 - `yarn berry`는 `node_modules`를 사용하지 않는 구조를 채택했다.
 - 파일을 찾는 데 어렵다.
- `nx`
 - `yarn berry` 단점 보완
 - 다양한 기능을 써야할 때 편리하다.
 - 특정 패키지만 배포해야 할 때 필요한 모듈만 정리해서 제공해준다.