당신의 Breaking Change는
누구의 시간을 태우나요?

혹시 라이브러리를 업데이트했다가

갑자기 빌드가 깨진 경험 있으신가요?

`@yarnpkg/plugin-npm` -> `@yarnpkg/core` (peer dependency)

1. `@yarnpkg/core`에 `DurationUnit`이 추가됨
2. `@yarnpkg/plugin-npm`은 `@yarnpkg/core`의 `DurationUnit`을 사용

위 업데이트는 **Breaking Change 없이**
새로운 항목이 추가되었기 때문에 minor 버전업이 되었습니다

그런데 정말 Breaking Change가 없었을까요?

`@yarnpkg/plugin-npm`은 `@yarnpkg/core`에 의존합니다
`yarn dlx`는 `@yarnpkg/plugin-npm`에 의존합니다

그리고 저희 cli 라이브러리는 `@yarnpkg/core`에 의존했습니다

그리고 Breaking Change를 통해 고통을 겪은 또 다른 이야기

사내 라이브러리 통합 프로젝트

문제 상황

흩어진 라이브러리

주인 없이 무분별하게 관리되는 수십 개의 라이브러리를 하나로 통합하여야 합니다

전사적 영향 범위

이 수십 개의 라이브러리들은 토스 코어 그리고 계열사까지 사용되는 라이브러리들입니다

예측 불가능

수십 개의 라이브러리들의 빌드 환경, 노드 버전, 패키지 매니저 등이 다릅니다
통합시켰을 때 어떤 사이드 이펙트가 생길지 아무도 예측하지 못했습니다

그럼에도 해야했다

문제

- 서비스 개발자분들은 바쁘다
- 이 작업들을 요청드렸을 때 언제 완료될지 알 수 없다
- 이 라이브러리들은 아카이브되었기 때문에 더이상 사용하면 안 된다

해결 방안

우리 팀이 직접 발로 뛰어서 문제를 해결해드리자

(서비스 마이그레이션 PR을 들고 미팅을 잡고, PR이 머지되기 전까지 미팅을 끝내지 않는 방법)

그러나 현실적으로 백여 개가 넘는 서비스를 모두 손으로 마이그레이션하는 시간이 너무 많이 듭니다

단순히 import가 변경된 라이브러리부터, API 자체가 변경된 라이브러리까지
각각의 케이스를 모두 외운 뒤 변경하는 것도 불가능합니다

해결 방안을 위한 해결 방안

Codemod라는 도구는 딱 이런 상황을 해결하기 위해 만들어진 도구입니다

Code Modification의 줄임말로, 코드를 조작하기 위해 코드를 작성하는 기술입니다.
Facebook의 React, Vercel의 Next.js는 Breaking Change가 있을 때 Codemod를 제공함으로서 사용자의 부담을 줄여줍니다.

```
import React, { useMemo } from 'react';
```

```
ImportDeclaration {
  specifiers: [
    ImportDefaultSpecifier {
      local: Identifier {
        name: "React"
      }
    },
    ImportSpecifier {
      local: Identifier {
        name: "useMemo"
      }
    }
  ],
  source: Literal {
    value: "react"
  }
}
```

```
...
ImportSpecifier {
  local: Identifier {
    name: "useCallback"
  }
}
...
```

`ImportSpecifier`의 `useMemo`를 `useCallback`으로 변경하면

```
import React, { useCallback } from 'react';
```

실제 코드 역시 `useMemo`가 `useCallback`으로 변경됩니다

Import Migration Codemod

legacy import를 새로운 패키지로 통합

```
const REPLACEMENTS = new Map([
  ['legacy-lib/Foo', 'new-core'],
  ['legacy-lib/Bar', 'new-core'],
  ['legacy-ui/Baz', 'new-ui'],
]);

export function transform(root, { j, stats }) {
  root.find(j.ImportDeclaration).replaceWith(({ node }) => {
    // 교체 맵 기반으로 import 변환
    // 기존 import 정리 후 그룹화해 삽입
  });
}
```

API Upgrade Codemod

import 이동 + 시그니처 변경 + 본문 재작성

```
export function transform(root, { j }) {
  // 1. import 이동/리네임
  rewriteImports(root, j, new Map([
    ['mock-lib/setup', 'mock-lib/browser']
  ]));

  // 2. 핸들러 시그니처 변경
  rewriteHandlers(root, j, {
    from: '(req, res, ctx)',
    to: '({ request, params, cookies })',
  });

  // 3. 본문 재작성 (async 도입 필요)
}
```

결과는?

기간: 4 ~ 5 개월

변경된 서비스: 100+ 개

codemod로 자동화했음에도 많은 기간이 걸렸습니다

이 과정에서 학습한 Breaking Change의 유형

1. codemod를 제공하지 않는 Breaking Change
    - A 라이브러리 1 버전을 사용하는 서비스가 있지만, 그 라이브러리는 7 버전까지 나왔음
    - 1 -> 7 까지의 codemod는 제공하지 않아서 모든 케이스에 대해 codemod를 작성
2. codemod가 edge case를 대응하지 못하는 경우
    - 개발자가 생각한 방법을 벗어나는 방법으로 사용한 경우
    - codemod는 결국 패턴을 따라 수정하는 거기 때문에 이 패턴을 벗어나면 예외 발생
3. codemod로 대응이 불가능한 Breaking Change
    - codemod의 한계: 여러 파일에 걸쳐 바라볼 수 없음, 런타임 값을 추적할 수 없음

해결책 1

codemod는 필수로 제공되어야 합니다

- Breaking Change를 만들면 codemod를 함께 제공해야 합니다
- 사용자는 명령어 한 줄로 시간을 많이 아낄 수 있습니다

```
npx @your-lib/codemod migrate-v2
```

해결책 2

실제 서비스에서 먼저 검증하세요

codemod를 만들더라도 이를 사용하는 서비스에서 제대로 동작하지 않으면 의미가 없습니다

실제 라이브러리를 사용하는 서비스를 대상으로 테스트를 해야 합니다

해결책 3

codemod로 안 되는 변경은 최대한 피해야 합니다

Deprecated 패턴, Feature Flag 패턴 등을 적극 활용해 사용자가 Breaking Change로 인해 코드를 직접 수정해야 하는 상황을 피하세요

결국 이런 방법들이 서비스 모노레포를 잘 관리하는 방법

Q&A