腐敗防止層による
スムーズなライブラリ移行

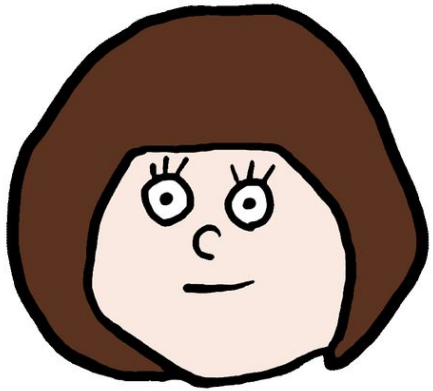
2024/8/24 フロントエンドカンファレンス北海道
@yoshiko

_KNOWLEDGE WORK

よしこ @yoshiko_pg

株式会社ナレッジワークのフロントエンドエンジニア
GUIをSPAとして作るのが好き。

自社での技術スタックや設計をZennで公開しています。
https://zenn.dev/yoshiko



目次

- 腐敗防止層とは？
- ベストプラクティスなのか？
- 実例の紹介
- 設けるのか、設けないのか

腐敗防止層とは？

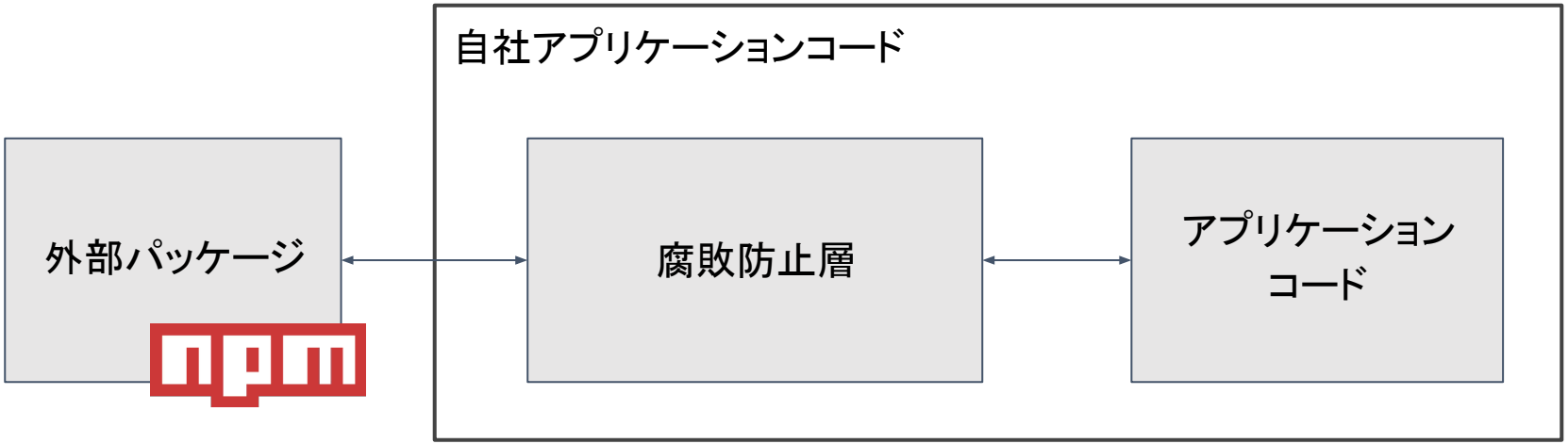
腐敗防止層の出典

出典：エリック・エヴァンスのドメイン駆動設計

ざっくりと：
システムAとシステムBの概念やモデルやインターフェイスが異なる場合でも
間に腐敗防止層というレイヤーを設けて相互に変換することで
互いを直接解釈する必要がなくなる

今回話す腐敗防止層のイメージ

原典に忠実にというよりもっと広義にコンセプトを踏襲したもの



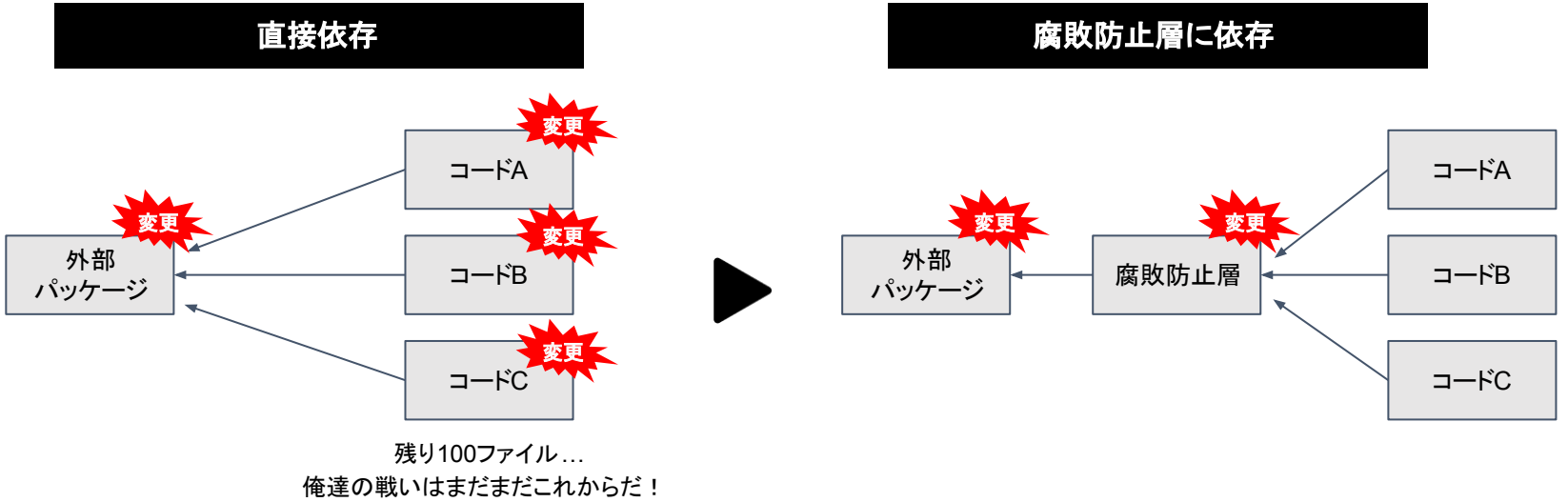
外部パッケージに対する腐敗防止層の効能①

- 外部パッケージは進化し続ける
- インターフェイスも変わる可能性がある

外部パッケージが更新され続けることは、利用側としても嬉しいこと

しかし、アップデートが入るたびにアプリケーションコードへ影響が出るのは嬉しくない
それを避けるためにバージョンアップができなくなってしまうのはもっと嬉しくない

外部パッケージに対する腐敗防止層の効能①



インターフェイスの変更を一箇所に閉じ込めて吸収できる

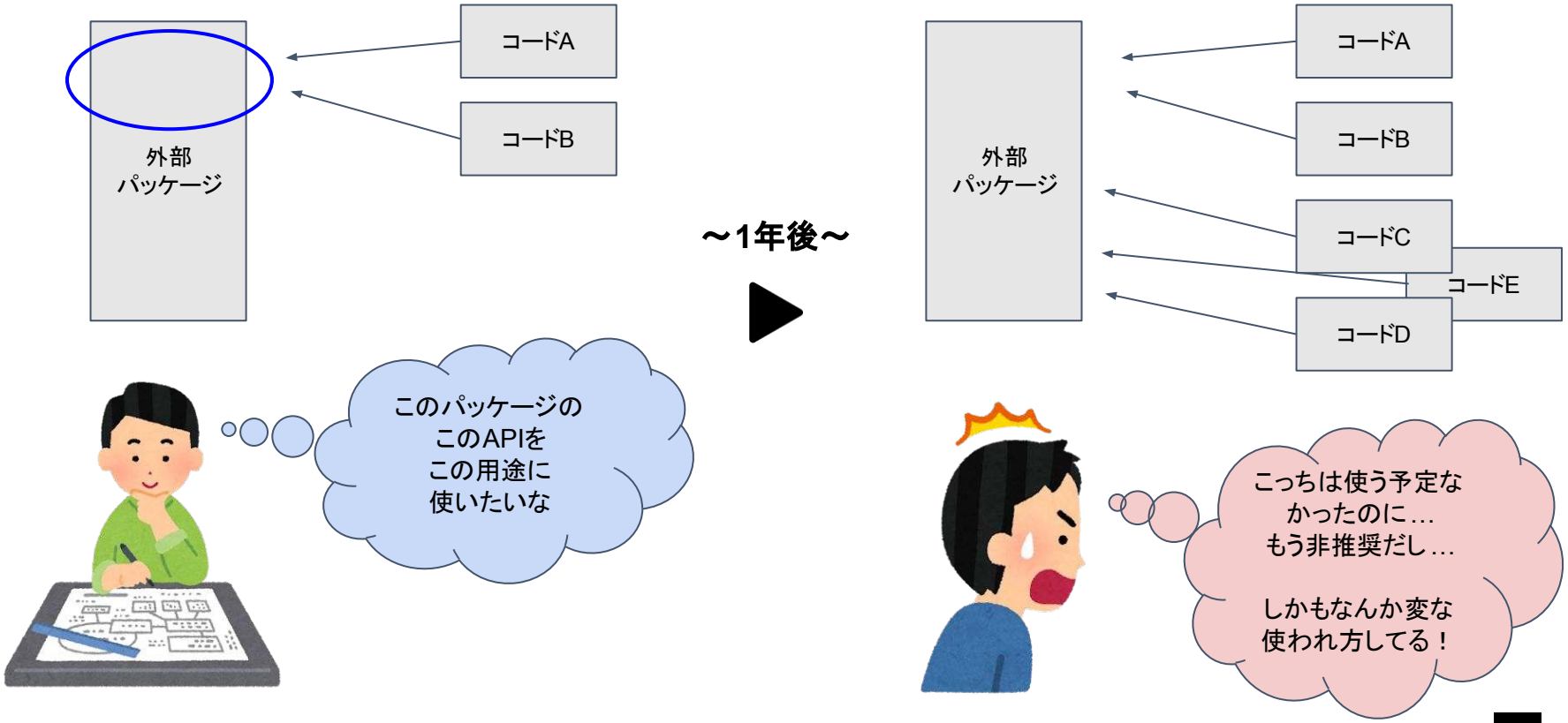
外部パッケージに対する腐敗防止層の効能②

利用している特定の外部パッケージの中にも、使いたい機能と使いたくない機能がある場合

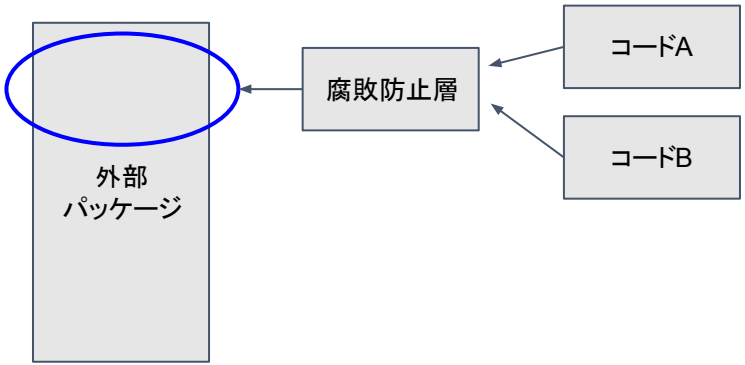
公開する範囲を制限できる

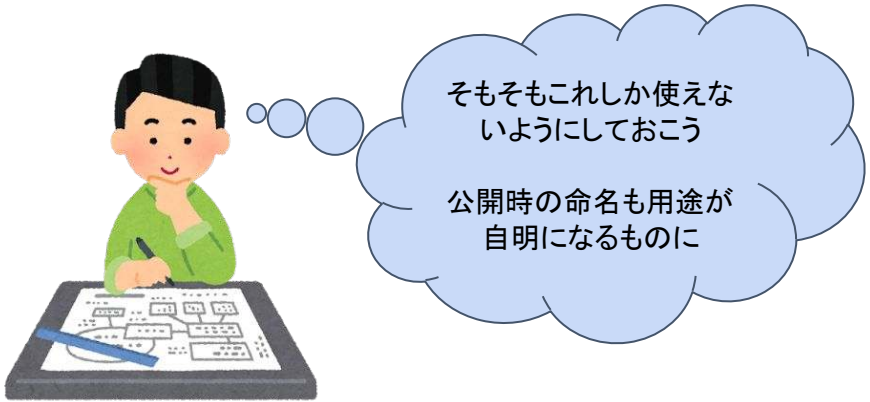
※ 使いたくない機能 ＝ 適切に使うことが難しい、何らかのリスクがある、既に同じ役割のものがある、非推奨化etc

外部パッケージに対する腐敗防止層の効能②



外部パッケージに対する腐敗防止層の効能②





そのパッケージの使われ方を当初の設計者がある程度コントロールできる

こちらのメリットは出典の「腐敗防止層」の意図よりは広そう

ベストプラクティスなのか？

ベストプラクティスではなくて
トレードオフ

だと考えています

外部パッケージに対する腐敗防止層はトレードオフの一選択

そのまま使う

- 外部パッケージを深く使う
- フル機能を自由に活用できる
- 実装の乗り換えはしづらい

腐敗防止層を設ける

- 外部パッケージを浅く使う
- 実装を乗り換えやすくする
- フル機能の活用はしづらい

どちらをとるのか？

右をとった場合、今回紹介したようなアプローチになる

実例の紹介

実例①　SWR → TanStack Query

- 元々SWRのuseSWRだけをexportして利用
  - かつ、optionは独自のインターフェイスを作り、腐敗防止層内で変換して利用
- 途中からTanStack Queryにしかない挙動（staleTime: 0）が必要になり、乗り換えた。
  - 実装の変更は腐敗防止層1ファイルのみの変更で完了

コードイメージ（ option部分）

Fetch Optionも生Optionだと毎回ドキュメントを見ないと適切に利用することが難しいが、
公開するインターフェイスをユースケースに沿ったものに簡略化しておくことで
使う側の知識も最低限で済む。

```ts
// アプリケーション内へ露出させるオプション
export type SWROptions = Partial<{
  /**
   * ポーリング間隔の秒数。1以上だとその間隔で再取得/画面反映を繰り返す
   */
  pollingSeconds: number

  /**
   * 一度だけ取得し、以降自動再取得しない。既にcacheがある場合は利用する
   */
  once: boolean

  /**
   * フォームの初期値取得用。
   * 既にcacheがある場合にもcacheを利用せずに取得し、以降自動再取得しない。
   */
  form: boolean

  /**
   * データがあった場合、新しいデータを再取得しない
   */
  preventRefetchWhenDataFound: boolean
}>
```

```ts
const createRawOptions = <Data>(opt: SWROptions) => {
  const options: Omit<QueryObserverOptions<Data>, 'queryKey'> = {}

  if (opt.pollingSeconds) {
    options.refetchInterval = opt.pollingSeconds * 1000
  }

  if (opt.once || opt.form) {
    options.refetchOnWindowFocus = false
    options.refetchOnMount = false
    options.refetchOnReconnect = false
  }

  if (opt.form) {
    options.gcTime = 0
  }

  if (opt.preventRefetchWhenDataFound) {
    options.staleTime = Infinity
  }

  return options
}
```

実例② Recoil → Jotai

- 元々Recoilの一部APIだけをexportして利用
- メンテナンスが止まってしまったため、Jotaiに移行
  - インターフェイスの差異は腐敗防止層の1ファイルの変更のみで吸収できた。

実例② Recoil → Jotai 学び

ただ実際に動かしてみると、インターフェイスが合っていてもライフサイクルや挙動に
微妙な差異があり、一部エッジケースでの挙動が異なってしまった。

学び:

実装を乗り換える際、腐敗防止層があればインターフェイスの差異の吸収は容易になるが、
それだけで実際の挙動の差異も吸収できるわけではないので要注意

設けるのか、設けないのか

外部パッケージに対する腐敗防止層はトレードオフの一選択だという話をしました。

選択するときに参考になりそうな点を最後に紹介します。

メリットが大きい状況:

- そのパッケージの領域が枯れておらず、今選定したものを使い続けるかわからない
- チームメンバーの成熟度にばらつきがあり、使われ方を一定コントロールしたい

メリットが小さい状況:

- そのパッケージを長く使いそうで、深くAPIを使っていくことのメリットが大きい
- チームメンバーが成熟しており、使われ方をコントロールする必要がない

弊チームでの需要の変化

4年前:

- 立ち上げフェーズで不確定要素が多かった
- 腐敗防止層を設けておいたことで状況変化にスムーズに対応できた

現在:

- 1→10 フェーズに入り、構成も固まってきた
- 外部パッケージのAPIをフル活用する需要の高まり
- 適切な判断ができるシニアなチームメンバーが増えている

だんだん要らなくなってくるフェーズに入ってきたなと感じてます。To Be Continued…

https://knowledgework.connpass.com/event/327800/ #encraft

東京虎ノ門でオフラインです！オンライン配信もあります！

できる喜びが巡る日々を届ける

Deliver the joy of enablement