ナレッジワークでの State管理とRecoil 活用事例

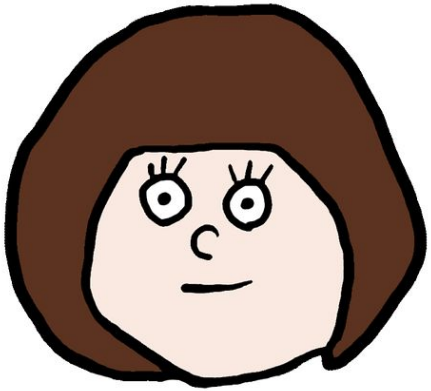
2023/01/20 Harajuku.ts Meetup

@yoshiko

_KNOWLEDGE WORK

自己紹介

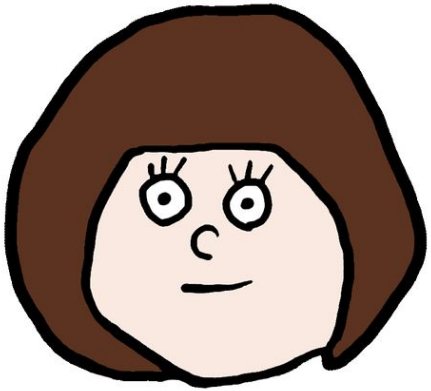
よしこ @yoshiko_pg



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

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

今日はこの話をじっくりします！

今日話すこと

- 前提：SPAにおける3種類のState
- Recoilを使ったGlobal Stateの運用ルール
- Recoilを使ったGlobal Stateのデータフロー

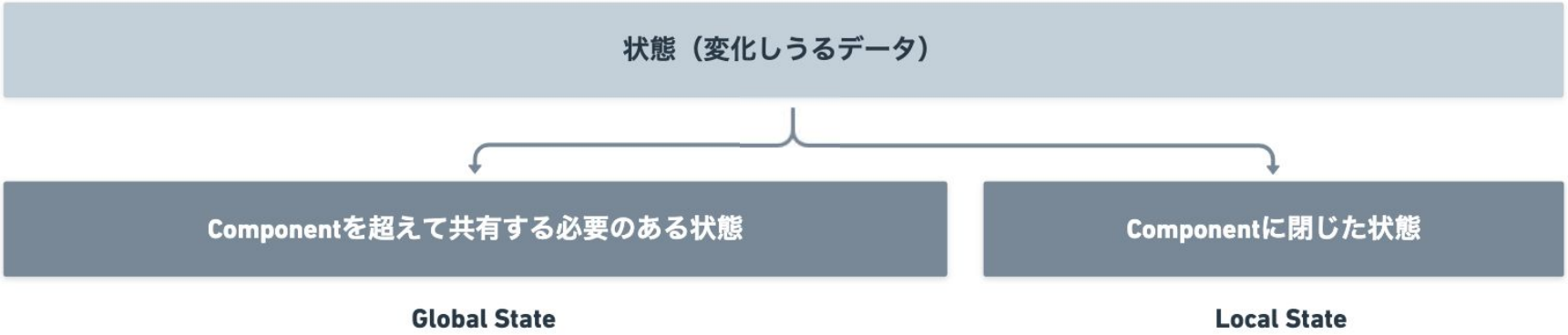
前提：SPAにおける3種類のState

Stateの分類

状態（変化しうるデータ）

State

Stateの分類



※ Component = Root Component以外の任意のComponent

※ 超えて = 他のComponentへだったり、ライフサイクルを超えて自身へだったり

Root ComponentがアプリケーションのGlobal空間であり、Global StateはそのGlobal空間のLocal Stateともいえるかも

ここでいうGlobal Stateはそれを指しているので、URLやLocal Storageなどの永続する値は含んでいません

Stateの分類

※ Component = Root Component以外の任意のComponent

※ 超えて = 他のComponentへだったり、ライフサイクルを超えて自身へだったり

Root ComponentがアプリケーションのGlobal空間であり、Global StateはそのGlobal空間のLocal Stateともいえるかも

ここでいうGlobal Stateはそれを指しているので、URLやLocal Storageなどの永続する値は含んでいません

Global Stateについて考える

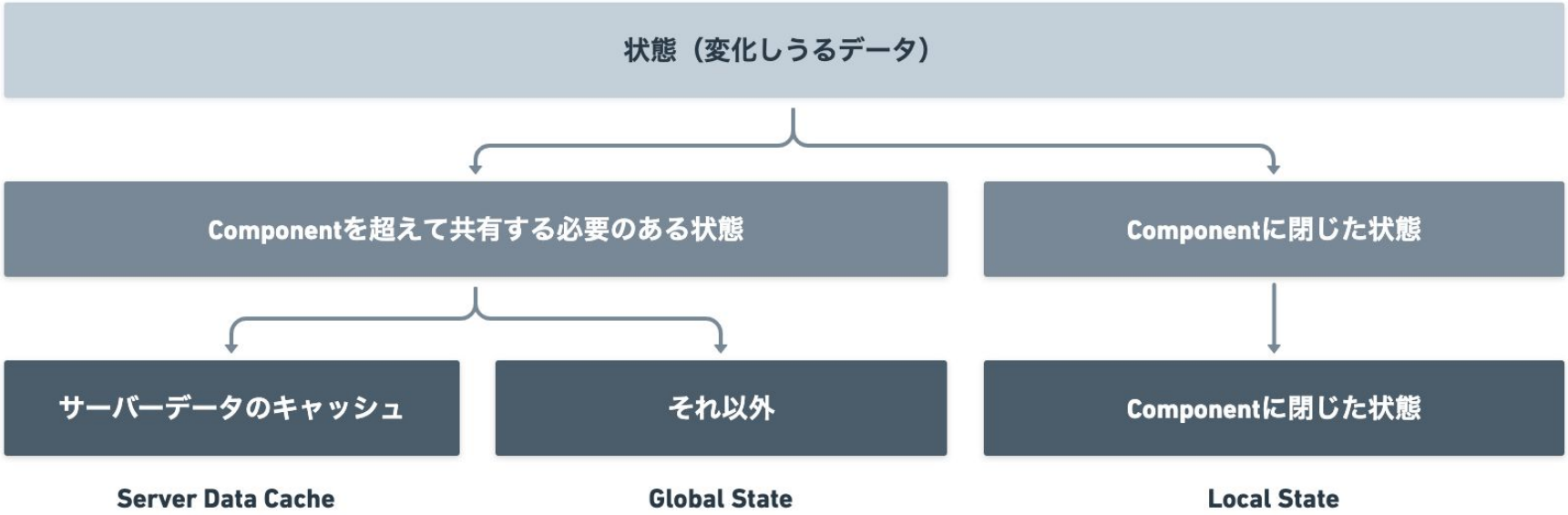
- Global Stateに置かなければならないデータって何だろう
  - だいたいはサーバーデータのキャッシュでは？
- 従来のGlobal Stateからサーバーデータのキャッシュを除くと、残るデータは僅か
- であればそれらは必要最低限の仕組みでそれぞれに管理すればよいのでは？

→ Global Stateをさらに分類してみよう

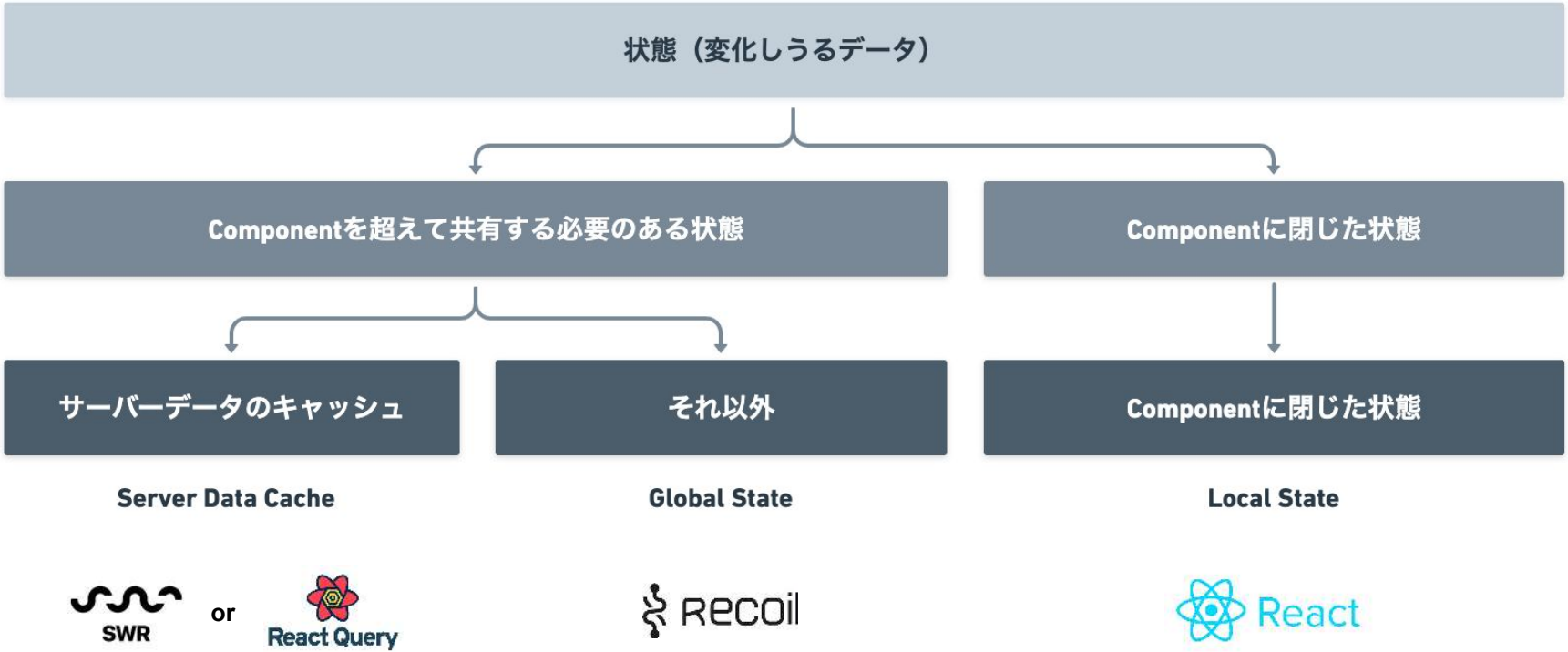
参考にしたツイート
（両方2020年2月頃）

Stateの分類

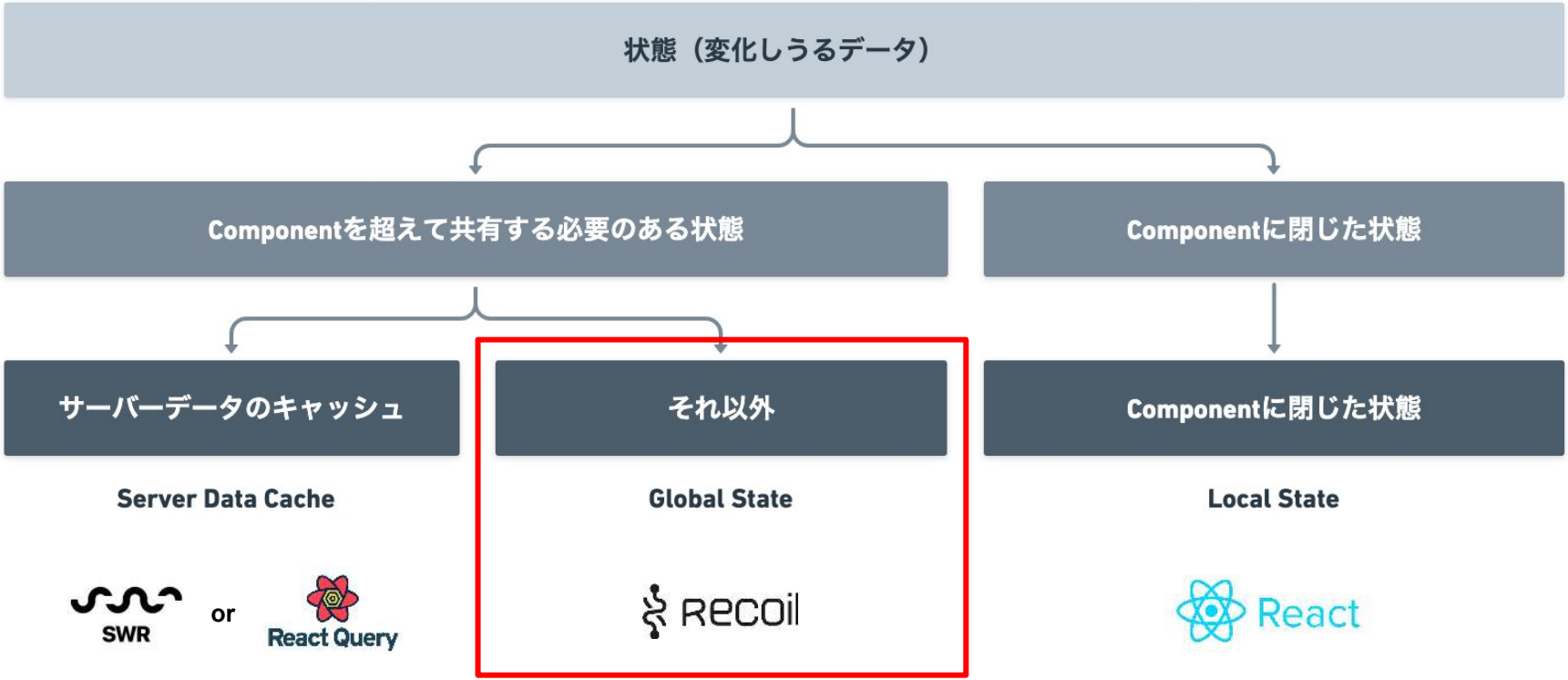
Stateの分類 - 最終形



Stateの分類 - 最終形



Stateの分類 - 最終形



Recoilを使ったGlobal Stateの運用ルール

今回紹介するナレッジワークでの Recoilの使い方は、
ピュアなGlobal Stateとしてのかなりシンプルな使い方になります。
APIも9割方atomしか使っていません。

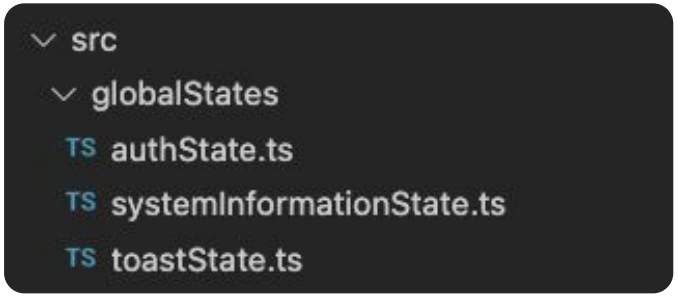
前段でStateを3つに分類したことで Global Stateの担う役割が最小限になり、
最大限シンプルな使い方でもニーズを十分カバーできるようになりました。

RecoilをつかったGlobal Stateの運用ルール

- ① 置き場所・命名のルール
- ② 露出させるインターフェイスのルール

① 置き場所・命名のルール

- src/globalStates 以下に
  1Stateあたり1ファイルを作成する
- ファイル名は xxxState.ts
  → ファイル名単体で見たときも役割が明確



- ファイル名をRecoilのkeyにする
  → keyが衝突しないことを担保できる

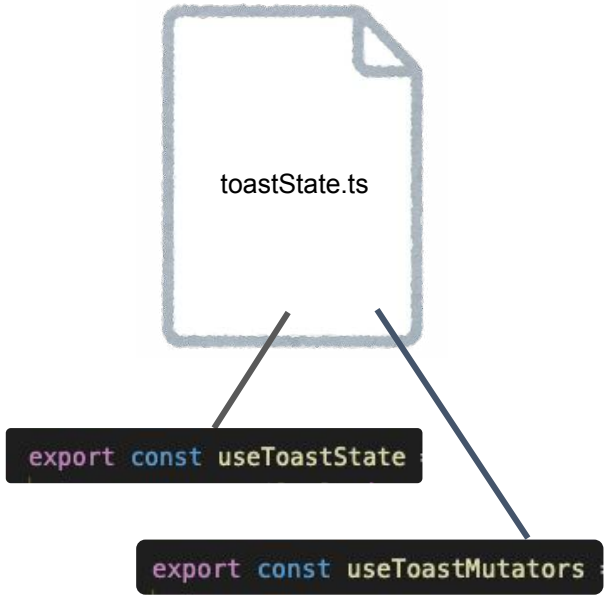
```ts
type ToastState = {
  toasts: Toast[]
}

const toastRecoilState = atom<ToastState>({
  key: 'toastState',
  default: {
    toasts: [],
  },
})
```

② 露出させるインターフェイスのルール

- StateそのものやsetStateなどRecoilのAPIを直接露出させず、Read用/Write用のカスタムフックひとつずつのみ露出させる
  - ライブラリの知識を隠蔽できる。腐敗防止層にもなる
  - （主にWriteで）操作に “意味” を付与できる

→ 今後Global State管理ライブラリを乗り換えたくなったときも、このGlobal Stateレイヤーのファイルと Root ComponentのProviderだけ書き換えればOK



exportするのはhooksふたつだけ！

Read hookのルール

- State の Read hook は use◯◯State の命名規則で export
  - だいたいuseRecoilValueをラップしてるだけ
- 任意のComponent / Usecaseから利用可能

```
export const useToastState = () => {
  return useRecoilValue(toastRecoilState)
}
```

Write hookのルール

- State の Write hook は use◯◯Mutators として export
  - setStateをラップしたWrite関数のまとまりを返す。setState は直接露出させない
  - Write関数は必ずメモ化する（使う側で depsに入れることを想定）
- 任意のUsecaseから利用可能

setStateを露出させるのではなく、行いたい
操作ごとに専用の関数を提供することで、
このStateはどんな操作を想定しているのか？
も伝えることができます

```
export const useToastMutators = () => {
  const setState = useSetRecoilState(toastRecoilState)

  const showToast = React.useCallback(
    (toast: Toast) => {
      setState(({ toasts }) => ({ toasts: [...toasts, toast] }))
    },
    [setState],
  )

  const clearAll = React.useCallback(() => {
    setState(() => ({ toasts: [] }))
  }, [setState])

  return { showToast, clearAll }
}
```

Callback hookのルール

- useRecoilCallback的な挙動を提供する場合は use◯◯Callbacks として export
  - あんまりないけど
  - Write hookと同じようにこちらも行いたいこと別の関数のまとまりを返す
- 呼び出しはReadと同じで任意のComponent/Usecaseから利用可能

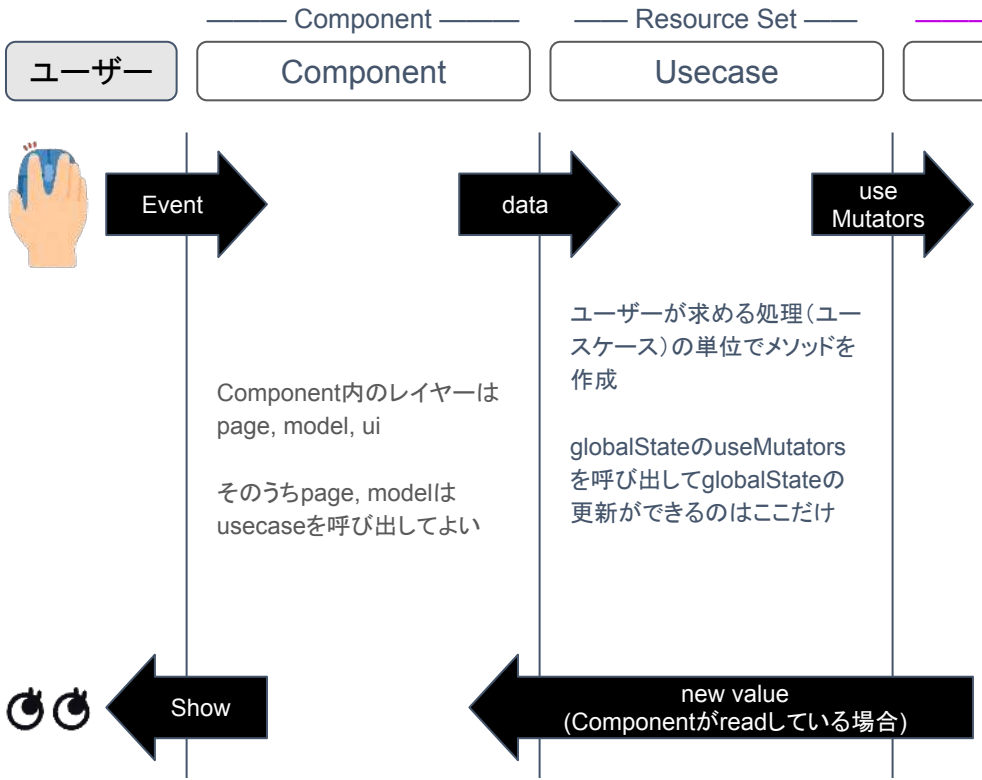
あまり使ってないけど、気の利いたAPIだなぁと思って個人的にはすごく好きです！

useRecoilCallbackを知りたい人はuhyoさんのブログを読もう！

“これは、Atomへのsubscribeは発生させたくないけど Atomの値を読みたいという贅沢な悩みを解決してくれるフックです。 ”

Recoilを使ったGlobal Stateのデータフロー

Global Stateの変更フロー



Resource Set/Usecaseって何？知りたい！という方はぜひ以下の記事を！

https://zenn.dev/yoshiko/articles/91a3dd575f99a2

おやくそく

We’re Hiring!!!

_KNOWLEDGE WORK

フロントエンドエンジニア

こんなお悩みのあなたにぜひ！

「気付いたらチームの中では
フロントエンド一番わかる人、
リードする人になっちゃったけど、
自分もまだ成長したいし
詳しい人のレビューも受けてみたいし
このままでい続けていいのか不安 …」

シニアフロントエンドエンジニア

明確な強みがあって、
この分野なら俺に任せろ！と言える人

自分の強みの分野において設計だけでなく
中長期的な戦略を立て実行できる人

（高い水準でジェネラルになんでも
できる、というのもひとつの強み）

https://kwork.studio/recruit-engineer

yoshikoにTwitter DM or 話しかけてもらうのも大歓迎！

おまけ: Why not?

Why not Context? - Zennの記事より

要件的にはReactに元々入っているContextでも実現できないことはないのですが、それを使わなかった理由は、APIの好みによる部分が大きいです。

- Stateを分割したいときにJSXとしてProviderを差し込む必要がある
- State更新の手段が提供されていないので、Setterも自分で用意して値と同列に生やす必要がある
- createContextとProviderのvalueの2箇所で初期値を指定しないといけない

あたりの使い勝手があまりしっくりきておらず、さらに今回は Single State + Selector という設計ではなくStateの用途単位で個別にStateを作っていく設計にしたいと思っていたので、使い勝手のよい形で Stateを柔軟に増減させられることが理想でした。
そこで、ちょうどその頃公開されて間もなかった RecoilのAPIがとても理想的だったため、採用を決めて開発を進めてきました。

Why not Redux?

- ReduxとSingle Stateに恩恵はあったか？
  - View(State) = UI の世界は綺麗だしテストで便利なのはわかるけど、関係のない stateがずらっと並び、延々バケツリレーしていく世界が普段の開発上嬉しいと思えない
    - 処理中のloading表示するごとに1セット用意し、error message表示するごとに…
- Single Stateなことで、大半の Stateを使わないページでも全 Stateのためのコードが必要になる
  - バンドルサイズにおける弱み
  - 初期化にも時間がかかる
- ボイラープレート多すぎ問題！ → Redux Toolkitで結構よくなったかも
- サーバーキャッシュ分離できるなら too much → むしろRTK Queryで一元化できるみたい

できる喜びが巡る日々を届ける

Deliver the joy of enablement