Suspense Fetchを3年実用してみて

2023/06/29 Encraft #4

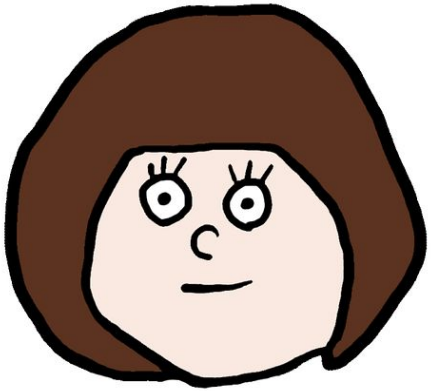
@yoshiko

_KNOWLEDGE WORK

ナレッジワークでは 2020年の創業時から
データフェッチに Suspense Fetchを利用してきました。

Suspense Fetchを利用することでコードはどう変わるのか、
また初期から使ってみた感想などを振り返ります。

よしこ @yoshiko_pg



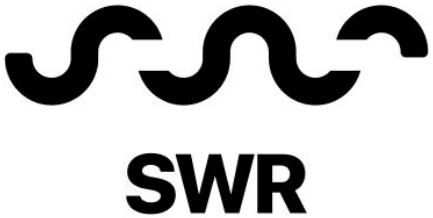
株式会社ナレッジワークのフロントエンドエンジニア

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

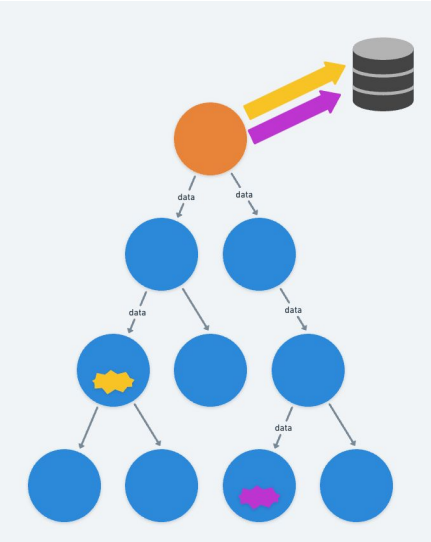
前段：近年の fetchスタイルの変化

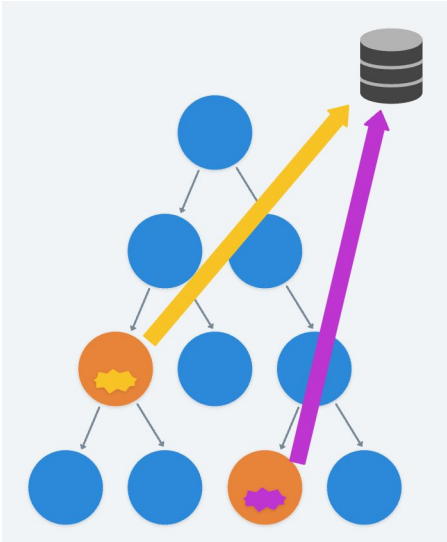


CacheつきFetching libraryの台頭

Page Containerから一括でfetch

各Componentからそれぞれにfetch





Fetching libraryが同タイミングでの同じ fetchをまとめてくれるので
各Componentが自分の関心のあるデータを取りに行けるようになりました

useEffectでのfetch

```tsx
const UserList = () => {
  const [users, setUsers] = useState<User[]>([])

  useEffect(() => {
    getUsers().then((data) => setUsers(data))
  }, [])

  return <p>{users.length}</p>
}
```

Fetching libraryでのfetch

```tsx
const UserList = () => {
  const { data, error, isLoading } = useSWR(
    key,
    getUsers,
  )

  return <p>{data && data.length}</p>
}
```

useEffectを使わなくても fetchに最適化された APIで
fetch処理とキャッシュをおこなえるようになりました

Suspenseによる変化

Suspenseによる変化（with SWR）

suspense: false

```
const UserList = () => {
  const { data, error, isLoading } = useSWR(
    key,
    getUsers,
  )

  return <p>{data && data.length}</p>
}
```

suspense: true

```
const UserList = () => {
  const { data } = useSWR(
    key,
    getUsers,
    { suspense: true }
  )

  return <p>{data.length}</p>
}
```

- isLoadingとerrorの状態がComponentの外に追い出された
- dataの型が T | undefined ではなく T になった

isLoadingとerrorの状態がComponentの外に追い出された


```tsx
const Page = () => {
  return (
    <ErrorBoundary>
      <Suspense>
        <UserList />
      </Suspense>
    </ErrorBoundary>
  )
```

```tsx
const Page = () => {
  return (
    <ErrorBoundary
      fallback={<p>Something went wrong</p>}
    >
      <Suspense fallback={<LoadingSpinner />}>
        <UserList />
      </Suspense>
      <Suspense fallback={null}>
        <SubInfo />
      </Suspense>
    </ErrorBoundary>
  )
```

- Loading時/Error時の表示領域を JSXのネストで柔軟に指定可能
- Loading時/Error時の表示内容を Wrapperのfallback propsで指定可能

dataの型が T | undefined ではなく T になった

_KNOWLEDGE WORK

T | undefined

```
const getData = () => {
  // data: Data | undefined
  const data = getDataFromOther()

  // data: Data | undefined
  return data
}
```

T

```
const getData = () => {
  // data: Data | undefined
  const data = getDataFromOther()

  if (!data) throw new Error()

  // data: Data
  return data
}
```

- 関数内で例外を発生させることで以降の処理は正常系に集中できるのと同じ姿

他フレームワークでの例

Vue

```
<Suspense>
  <UserList />

  <template #fallback>
    loading...
  </template>
</Suspense>
```

Vue.js 3.0でSuspenseが実験的機能として登場
ErrorBoundary相当はまだ存在しない

Angular

```
@Component({
 selector: 'app',
 template: `
    {#defer on viewport}
      <calendar-cmp
       ...
      />
    {:loading}
      Loading...
    {:error}
      Error while loading
    {/#defer}
 `,
})
class App { }
```

ngconf 2023で関心の近いRFCが発表された
画像引用元Tweet(Angular開発者の方)

使ってみてどうだったか？

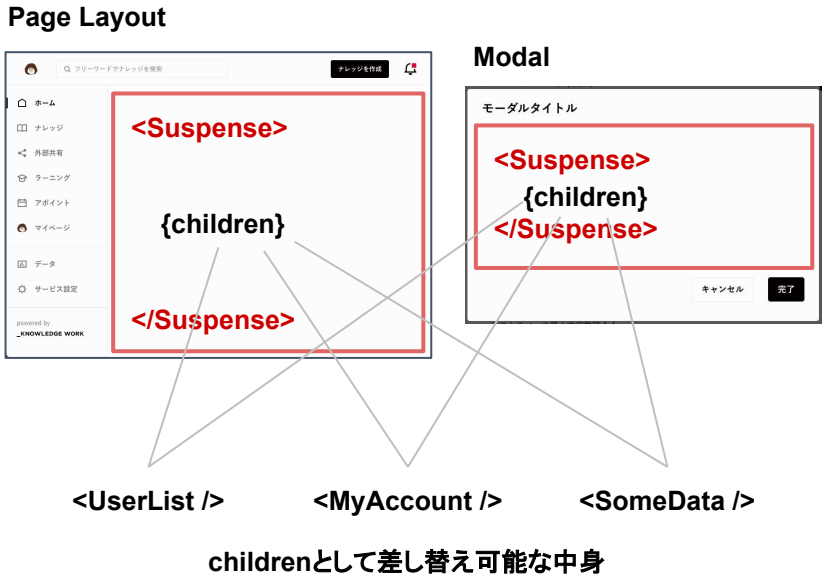
良かったこと

書きやすさ、コードの見通しの良さ

- 非同期処理をPromiseではなくasync/awaitで書けるようになったときの感動と同じ感動
- loadingとError処理をComponentの外側に追い出せることにより Component内部がシンプルに
- loadingとError処理の境界をJSXで自由に設定できる柔軟性
    - ここは一緒にしたい、ここは分割したい、などが思いのまま

差し替え可能なローディングコンテナとコンテンツ

- WrapperとしてLoading/Errorをハンドリングするコンテナ層と、Fetchをする実体であるコンテンツ層を分離できることで、これらが相互に入れ替え可能になる
  - 例えばページレイアウトのメイン部分やModalの外枠にErrorBoundary/Suspenseまで含めておくと、内側でどんなComponentがFetchしても同じようにLoading/Error表示ができる



- React Server ComponentsもSuspenseベースの設計なので、移行も楽かも？
  - とはいえFetchをServerに移すかどうかは、認証周りの前提が大幅に変わったり
    オフライン対応どうする？とかもあるので、慎重に考えている
  - Serverに移す場合はfetching libraryを通さずRSC用fetchでやりそう

使ってみてどうだったか？

課題

waterfall問題

- Suspense Fetchを同じComponent内で2つ以上呼ぶと直列実行になってしまう（ウォーターフォール問題）
- ナレッジワークでは read() を挟むインターフェイスに拡張してfetchとsuspenseのタイミングを分離している

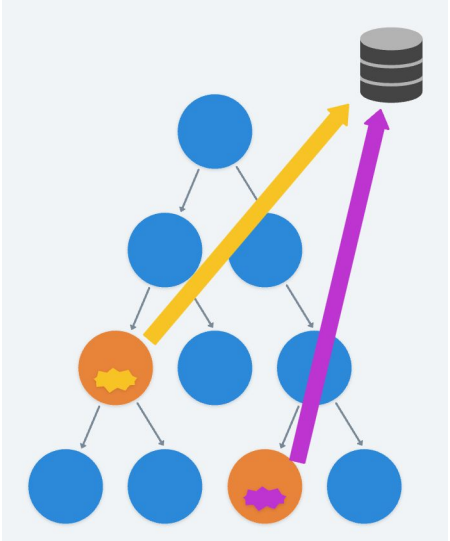
```
const Page = () => {
  const { data: user } = useSWR('/api/user')
  const { data: movies } = useSWR('/api/movies')

  return <p>{user.name} {movies.length}</p>
}
```

```
const Page = () => {
  const userResource = useSWR('/api/user')
  const moviesResource = useSWR('/api/movies')

  const user = userResource.read()
  const movies = moviesResource.read()

  return <p>{user.name} {movies.length}</p>
}
```

- SWRでもこの問題へのissueやPRが立っているが、まだどれも着地していない。整理gist
- 解決案として個人的に一番期待しているのは React RFCの use hook
  - Promiseを渡すとSuspenseするというI/Fなので、Promiseを作るために必然的に事前にfetchが走る
  - uhyoさんのuse解説記事がとても詳細かつ面白いのでおすすめ > 最速攻略！Reactの `use` RFC

fetch-on-render問題

- 現状はComponentがrenderされたタイミングでデータを取得するfetch-on-renderの方法になっている
- より良いrender-as-you-fetch（既に取得したデータを描画）のやりかたは、ReactでもSWRでもTanstack Queryでもまだ主流な方法がない
    - 事前にprefetchするとか、上部でfetchだけした未解決のresourceを子Componentにpropsで渡すとかになる
    - 親子間でFetchの関心を分離する方針と逆行するので、ナレッジワークではどちらの方法も現状とっていない
    - 右図の方針だと必然的にfetch-on-renderにならざるを得ない
- 正直ここまで長く主流な方法が出てこないとは思っていなかった
- そもそもライブラリでのSuspense Data Fetchingは、experimentalではなくなったもののnot recommendedではあるようなので仕方ないかも



使ってみてどうだったか？

総括

総括

- Componentと非同期処理を繋ぐデザインとして、とても優れたインターフェイスだと感じている
  - 他のフレームワークにも輸入が進んでいることからも支持が見て取れる
  - Suspense/ErrorBoundaryの振る舞いを含んだLayout Component/UI Componentを作れるのは大きい
- experimental時代から使っているが、目立ったトラブルや不具合は踏まなかった
  - とはいえ公式には未だNot Recommendedなので実際の利用は手放しにおすすめはしない
  - 概念を把握しておくのは間違いなくおすすめ
- Fetchタイミングの最適化とComponentの独立性にはトレードオフがある
  - 各Componentがそれぞれ自分に必要なdata fetchをするスタイルは楽だがタイミングが最適化されない
  - 親側でfetchまでおこないリソースを子に渡す型式ならタイミングは最適化されるが親子が密になる
  - ここのベストプラクティスはまだこれというものがない認識。useやRSCによっても変わってきそう

個人的には使ってよかったしこれからも使っていきます！
ご利用は計画的に

できる喜びが巡る日々を届ける

Deliver the joy of enablement