React 19 ハンズオン

アジェンダ

- 前提知識とおさらい
  - Reactのバージョン履歴
  - JSXとReact要素について
  - `Suspense`
  - トランジション
  - Reactの歴史: CSR,SSR,RSC
- React 19
  - アクションの導入: `<form>`, `useActionState`, `useFormStatus`, `useOptimistic`
  - `Suspense`と統合された新しいAPI `use`
  - サーバーコンポーネント
  - サーバーアクション

Reactのバージョン履歴

- 2019年2月 v16.8 hooksが登場
- 2020年10月 v17.0 新機能「なし」
- 2022年3月 v18.0 詳細はこちら React v18.0
  - 並行レンダリング
  - トランジション
  - `useId`, `useTransition`, `useDeferredValue`のフックを追加
- 2024年4月 v19-beta
  - サーバーコンポーネントやサーバーアクションなどhooks以来の大きな変化

前提知識: JSXとは何か

- JSXは`createElement`の糖衣構文
- コンパイラによって以下のように変換される

  厳密には別の関数に変換されます。詳細は新しい JSX トランスフォームを参照

```
import { createElement } from "react"

// jsx
<a href="https://example.com">Hello World!</p>

// transformed
createElement("a", { href: "https://example.com", children: "Hello World!" })
```

- では`createElement`は何をしているのか？

前提知識: React要素とは何か(1/2)

- `createElement`は名前の通り要素を作成している。
- `createElement(Greeting, { name: 'Taylor' })`の戻り値は以下のようなものになる

```
{
  type: Greeting,
  props: {
    name: 'Taylor'
  },
  key: null,
  ref: null,
}
```

前提知識: React要素とは何か(2/2)

- この戻り値のことをReact要素(React Element)と呼ぶ
- **つまりReactコンポーネントとはReact要素を返す関数であると言える**

  正確には文字列や数値なども返しても良いのですが、ここでその説明は割愛します。
- なお `createElement` を呼んだだけではReact要素が作成されるだけで、画面が変更されるといったことはない
  - `document.createElement` を呼んでも `appencChild` などを追加で呼ばなければDOMに反映されないのと同じ
  - 参考: React 要素とは要するに何なのか？
- 実際のソースコード読んでみるとそこまで複雑なことしてないのが分かります。
  ReactJSXElement.js#L775

前提知識: レンダリングとは何か

- Reactの世界では以下のような用語を使う
  - **レンダリング**あるいは**レンダー**とはReactコンポーネントを呼び出すこと
  - **コミット**とはReact要素を元にDOMを更新すること
  - **ペイント**とはDOMを元に実際の画面を描画すること
- 参考: レンダーとコミット

前提知識: Suspenseのおさらい(1/3)

- Suspenseとは読み込み中にフォールバックを表示できる機能のこと
- これによりローディングなどの状態を宣言的に書ける
- 以下のような形で使う

```
<Suspense fallback={<Loading />}>
  <MainContent>
</Suspense>
```

- `MainContent`が読み込み中で表示できない場合、代わりに`<Loading />`を表示する
- `MainContent`の読み込みが終わると、再度`MainContent`がレンダリングされる
- では読み込み中とは一体何を指しているのか？

前提知識: Suspenseのおさらい(2/3)

- 読み込み中の表現はJavaScript組み込みの `Promise` を使用している
- `Promise` をthrowすることでレンダリングを中断できるようにしている
- `Promise` をthrowされると `Suspense` がそれをキャッチしてフォールバックします
- キャッチした `Promise` が解決されたら再度レンダリングを試行するという流れ
- ただし `Promise` をthrowするというのは内部実装で公式ドキュメントには書かれていない
- アプリケーションコードでは `Promise` をthrowするコードは書かず、ライブラリやフレームワークの機能を使うべき

前提知識: Suspenseのおさらい(3/3)

- Reactの組み込みの機能には`Promise`をthrowせずに`Suspense`と連携してコンポーネントをサスペンドさせる機能をもつものもあります。
  - 後述のトランジションや非同期コンポーネントなど
- 参考リンク
  - ReactのSuspenseで非同期処理を乗りこなす
  - Promiseをthrowするのはなぜ天才的デザインなのか #JavaScript - Qiita
  - Suspenseはどのような機能なのか｜ReactのSuspense対応非同期処理を手書きするハンズオン

前提知識: トランジション(1/2)

- トランジションとは段階的推移を行うReact18からの新しい機能
- `setState`などによる更新を緊急性の高いものと低いものに分ける
  - 従来の更新はすべて非トランジション（緊急性が高い）
- トランジションを使うと以下のようなことができる
  - レンダリングを中断する
  - レンダリング中に処理の割り込みをする
  - `Suspense`と連携して表示を遅らせる - 公式デモ
- 使用方法はuseTransitionまたはstartTransitionのどちらかのAPIを使用する

前提知識: トランジション(2/2)

- 注意点
  - トランジションは同期関数しか渡せない(React18まで)
  - トランジション中はコンポーネントが並列に2度レンダリング(≠コミット)される
    - Reactがコンポーネントの純粋性を重視したり、StrictModeでコンポーネントをわざと2回レンダリングしたりする1つの理由
- 将来的にはライブラリやフレームワークのAPIで使われていくようになるので、アプリ開発者が直接使うことは少なくなっていくはず

Reactの歴史：CSR

- CSRとはClient Side Renderingの略
- Reactを使う場合、初回レスポンスでは空のHTMLを返し、その後ReactでDOMを組み立てるということが行われた
- ただし色々な問題点を抱えていた
  - 空のHTMLを返すので一瞬真っ白なページが表示されUXが良くない
  - APIを呼び出す場合、初回のレンダリングが終わってからAPI呼び出しをするので更に表示が遅くなる
  - クライアントからAPIを呼び出すと、OGPの動的生成ができない
    - ユーザー投稿型のようなサービスで問題になる
- これらの問題を解決するためSSR、SSGが生まれる

Reactの歴史：SSR、SSG

- それぞれ Server Side Rendering, Static Site Generation の略
- SSRとSSGはrenderToStringとhydrateRootといったAPIを使用する
  - `renderToString`でサーバーでレンダリング後HTMLを生成、`hydrateRoot`で再度レンダリングしてイベントハンドラをアタッチしている
- これでCSRの問題は一部解決したものの残った課題や別の課題が生まれた
  - クライアント側で不要なJSをダウンロードする必要がある
  - 初期データの取得を一度に一箇所に行う必要がある
  - 初期データの取得がフレームワークごとに異なる
- これらの問題を解決するためサーバーコンポーネントが生まれる

React 19 新機能

新API: use

- 新しく追加された特殊なフック
- `use(promise)` と呼び出すと引数の `Promise` が解決していた場合その戻り値を返す
- 保留中（pending）の場合、サスペンドする
  - つまり親の `Suspense` のfallbackが表示される
- これにより `Suspense` が簡単に扱えるようになる
- 後述のサーバーコンポーネントと組み合わせることで真価を発揮する

サーバーコンポーネント(1/2)

- React 19より新しく導入されたコンポーネントの新しい種類
- SCまたはReact Server Componentsの頭文字を取ってRSCと略される
- 従来のコンポーネントはクライアントコンポーネント(CC)と呼ぶようになった
- SCの最大の特徴は**サーバーサイドでのみ実行**されること
  - SSRはサーバーとクライアントの両方で実行していた
- CCとSCは排他ではなく併用できる
- サーバーサイドでのみ実行されるとはどういうことか？

サーバーコンポーネント(2/2)

- CCはReactコンポーネントを実行するとReact要素が返ってきていた
- SCとは単にサーバーでコンポーネントをレンダー(≠コミット)、その結果のReact要素をクライアントに返す機能と言える
  - レンダリングとは単にReact要素を作るだけの関数なので、コンポーネントによってはサーバーのみで実行しても問題ないはず
- CCの型を `(props: Props) => ReactElement` とするなら、SCは `(props: Props) => Promise<ReactElement>` と言い換えられる
- SSRとの違いは `renderToString` などはHTMLへと変換していてReact要素を返してはなかった
- こちらが分かりやすいです。[やっと React Server Components が腑に落ちた #JavaScript - Qiita]

サーバーコンポーネントを使うメリット

- Zero bundle size
  - 前述の通りサーバーで実行されてReact要素返すだけなので、SCはブラウザ側のJavaScriptが不要になる
- 非同期コンポーネントが利用可能になる。また`Suspense`とも深く統合されている
  - データ取得のコードが簡潔になる

サーバーコンポーネントの特徴(1/2)

- SCが有効な環境ではデフォルトで全てのコンポーネントがSCになる
- 逆にCCを使うにはファイルの先頭に `"use client";` というディレクティブを宣言する
- Promiseを返せる(=async/awaitが使える)。また `Suspense` と深く統合されている
  - `Suspense` の子要素がSCでPromiseを返した場合サスペンドする
- フックを使えない

サーバーコンポーネントの特徴(2/2)

- SCからCCをインポートできるが、CCからSCをインポートできない

  あくまでインポートの話で、DOMの親子関係の話ではないことに注意
- CCの呼び出し時にはシリアライズ可能な値しか渡せない (公式ドキュメント)
- CCの子要素にSCを渡すことはできる
  - CCからSCの呼び出しはできない
    - CCからSCのインポートが出来ず、またSCからCCに渡せるpropsはシリアライズ可能なもののみのため

レンダリングの流れ

- 初回リクエスト時
  1. SCをサーバーでレンダリング
  2. CCをサーバーでレンダリング
  3. CCをクライアントでレンダリング
- ページ遷移時
  1. SCをサーバーでレンダリング
  2. CCをクライアントでレンダリング
- SSRされるのは初回リクエスト時のみ
  - SSR用のAPIがHTMLを作成するだけと考えると自然な流れ

非同期トランジション

- React19から `startTransition` に非同期関数を渡せるようになった
- 非同期関数の戻り値のPromiseがresolveされるまで、isPendingはtrueのままとなる
- これにより非同期処理のローディング状態が扱いやすくなった
- またReact19からは非同期トランジションを使用する関数を規約として「アクション」と呼ぶ

`<form>`とアクション

- `<form>`の`action`属性に関数を渡せるようになった
  - HTMLの`action`属性は文字列しか渡せないのでこれはReact独自の拡張
- submitイベント時に`action`属性に渡した関数が実行される
- `action`属性に渡した関数は文字通り「アクション」(=非同期トランジション)となる
- `<button>`などにも新しく`formAction`属性が追加された
  - これは`<form>`の`action`属性をオーバーライドするような挙動となる
- 注意点としてアクションが成功すると非制御コンポーネントのフォームは入力内容がリセットされます。(公式ブログ、GitHubのissue)
- 後述の新しいフックと組み合わせることで真価を発揮する

useFormStatus

- `<form>`配下のコンポーネントで`useFormStatus`を使うことで、その`<form>`のトランジション中かどうかなどの状態を取得できる
  - `<form>`がコンテキストのような役割を果たす
- これにより送信中に`<button>`をdisabledにして二重送信を防止するなどの処理が書きやすくなった

useActionState

- アクションの結果をstateとして保持できるフック
- アクションでAPIを呼びレスポンスをstateに入れる場合、このフックを使うと便利
- `useState` と `useTranstion` を合わせたようなもの

useOptimistic

- optimisticとは楽観的という意味
- Optimistic Updates(楽観的更新)と呼ばれるUI/UXの実装方法(デザインパターンのようなもの)がある
- サーバーへ更新のリクエストを送りそのレスポンスをUIに反映する際に、レスポンスが来る前にUIを更新してしまうというもの
  - 即座にUIを更新した方がユーザーから見える待ち時間が少なくなる
  - リクエストが成功するだろうという前提でレスポンスを見ずに更新をするため、楽観的更新と呼ばれている
  - SNSのいいね機能などでよく使われている
- この `useOptimistic` フックを使うとそれを簡単に実現できる
- 具体的にはトランジション中だけstateを別の値に差し替えるといったことでできる

サーバーアクション

- 一言で言うとReact組み込みのRPC
- クライアントからサーバー上の関数を実行できる
- `"use server";` というディレクティブで宣言できる
- `"use server";` を宣言したファイルの関数、または1行目に `"use server";` と書いてある関数がサーバーアクションとなる
- サーバーアクションはシリアライズ可能(SCからCCへpropsとして渡せる)