全4回で学ぶNext.js入門講座
4回目

2025/11/25

Kazuma SEKIGUCHI

前回のアジェンダ

- SSRとは？
- サーバーによるレンダリングの仕組み
- サーバーコンポーネントでの fetch
- 外部APIからのデータ取得
- revalidate（キャッシュと再検証）の体験

今回のアジェンダ

- クライアント側データ取得（SWR）
- SSR＋SWRの組み合わせ
- レンダリング戦略の整理
- SSR / SSG / SWR の違いと使い分け
- DBへのアクセス・ServerActions

SWRとは

- SWR ＝ Stale While Revalidate（古いキャッシュを即返しつつ、裏で最新データに更新する）戦略を採用したデータフェッチライブラリ
  - Vercelが開発したReact用データ取得ライブラリ
  - Next.js公式推奨のクライアントデータ取得方式

SWRとは

- RSCはサーバー側でレンダリングをおこなう
  - 最新のデータを表示するためにはサーバー側の再実行が必要
  - 即座に最新状態にしたい場合には向いていない
  - リアルタイムでデータを変更する場合は、クライアント側で完結した方が高速に対応できる
- useEffectとfetchで対応
  - リアルタイムでのデータ取得は可能だが、キャッシュが存在しないため、毎回ローディングになってしまう
  - エラーハンドリングが煩雑
    - エラー時の処理が複雑になりやすい

SWRの利点

- SWRはさまざまな機能を提供してくれる
  - キャッシュ機能、裏で新しいデータを取得する、自動再取得、定期ポーリング、エラー管理、キャッシュの即時更新など

- 利用する時は、クライアントコンポーネントとして動作させる
  - swrをimportする

```
'use client'
import useSWR from 'swr'
```

SWRの動作

- キャッシュが存在していれば直ぐに返す
  - 裏で新しいデータを取得する
  - 最新データが取得できたら更新を行なう
- ブラウザーがアクティブになったときに更新する
  - オフラインがオンラインになったときにも更新する

- 初回はSSRで高速に動作し、2回目以降は自動更新が容易におこなえる点が強力
  - RSCは初回のレンダリングに必要
  - RSCが不要ということではない

レンダリング戦略

- SSR（Server-Side Rendering）
  - リクエストのたびにサーバーがHTMLを生成して返す
  - 毎回最新データを取得できる
  - HTMLが返って来るので、SEOに最適
  - 動作が重くなりがち
- SSG（Static Site Generation）
  - ビルド時にHTMLを生成して、生成したHTMLを返す
  - 高速に動作する
  - サーバーでの演算がほぼ不要になる
  - データが変わると再ビルドが必要になる

レンダリング戦略

- ISR（Incremental Static Regeneration）
  - SSGの弱点である更新困難を解決した仕組み
  - 決めた周期でHTMLを再生成する
- CSR（Client Side Rendering）
  - HTMLは空として返し、JSがクライアント側で動作してデータを取得して表示をおこなう
  - 初回の表示まで時間が掛かる
  - SEOに対して弱い
  - 常に最新データが表示できる

レンダリング戦略

- SWR（Stale-While-Revalidate）
  - クライアント側でキャッシュしつつ裏で最新データに更新する
  - リアルタイムに近い動作が可能
  - 初回SSRとして実行することが可能

  - 現状では一番良く使われる構成

レンダリング戦略（使い分け）

- 表示内容によって選択をおこなう
  - 常に最新のデータを表示する：SSR＋SWRかCSR＋SWR
  - ほぼデータに変化がない：SSG
  - たまにデータが変化する：ISR

- Next.jsで普通作成すればSSRになる
  - SWRを使うならSSRにプラスすればOK
  - SSGを使う場合は、use clientを使わないようにして作成する
  - CSRはuse clientだけで作成する

Next.jsからDBにアクセスする

- 現在のNext.jsではPrismaを使ってDBにアクセス、データ取得をするのが一般的
  - Prisma：ORMライブラリー
- さまざまなデータベースと連携することが可能
  - ライブラリーなので導入して、使えるようにしておく
  - DB接続用の設定は必要

Prismaでのデータ取得

```
npm install prisma@6 --save-dev
npm install @prisma/client@6
```

- prismaをnpm経由でインストールしておく

```
npx prisma init
```

- npxを使って初期化しておく
  - .envとprisma/schema.prismaが生成されるので変更する

```
DATABASE_URL="file:./dev.db"
```

.envの設定例（SQLite）

prisma/schema.prismaの設定

```
datasource db {
  provider = "mysql"  // or "postgresql" / "sqlite"
  url      = env("DATABASE_URL")
}

generator client {
  provider = "prisma-client-js"
}

model User {
  id    Int    @id @default(autoincrement())
  name  String
  email String  @unique
}
```

使用するデータベース種類を指定

テーブルについて記述

テーブルのフィールドについて記述する

prisma/schema.prismaの設定

- 設定をしたらコマンドを実行しておく

```
npx prisma migrate dev --name init
```

- 以降schemaを変更するたびに以下のコマンドを実行

```
npx prisma generate
```

データの取得、格納

```
const users = await prisma.user.findMany()

await prisma.user.create({ data: { name: "Taro" }})

await prisma.user.update({
  where: { id: 1 },
  data: { name: "Hanako" }
})

await prisma.user.delete({
  where: { id: 1 }
})

await prisma.user.findMany({
  where: { name: { contains: "a" } }
})
```

取得

作成・追加

更新

削除

条件付き取得
containsはLIKE文として機能する

条件式

```
const users = await prisma.user.findMany({
  where: { id: 1 }
})
```

単一条件

```
const users = await prisma.user.findMany({
  where: {
    name: "Taro",
    age: 20
  }
})
```

ANDでの条件

```
const users = await prisma.user.findMany({
  where: {
    OR: [
      { name: "Taro" },
      { age: 20 }
    ]
  }
})
```

ORでの条件

```
const users = await prisma.user.findMany({
  where: {
    name: {
      contains: "a"
    }
  }
})
```

文字列部分一致

```
const users = await prisma.user.findMany({
  where: {
    age: {
      gt: 20,  // greater than
      lt: 30   // less than
    }
  }
})
```

>と<条件
>=の場合はgteを使う

条件式

```
const users = await prisma.user.findMany({
  where: {
    NOT: {
      age: 20
    }
  }
})
```

NOT条件

```
const users = await prisma.user.findMany({
  where: {
    deletedAt: null
  }
})
```

NULLチェック

```
const users = await prisma.user.findMany({
  where: {
    deletedAt: { not: null }
  }
})
```

NULLではない

```
const users = await prisma.user.findMany({
  where: {
    age: { gt: 20 }
  },
  orderBy: {
    age: "desc"
  }
})
```

ORDER BY

```
const users = await prisma.user.findMany({
  where: {
    age: { gt: 18 }
  },
  take: 10,
  skip: 20
})
```

LIMITとOFFSET
LIMIT=take,OFFSET=skip

データの読み取り、保存

- 読み取り時はRSCを使うのが一般的

```
// Page.js (Server Component)
import { prisma } from "@/lib/prisma"

export default async function Page() {
  const users = await prisma.user.findMany()
  return <UserList users={users} />
}
```

- データ保存時はServerActionsを使う

```
"use server"
import { prisma } from "@/lib/prisma"

export async function addUser(formData) {
  await prisma.user.create({
    data: { name: formData.get("name") }
  })
}
```

サーバー側

クライアント側

```
<form action={addUser}>
  <input name="name" />
  <button>追加</button>
</form>
```

ServerActions

- APIを書かずにクライアントからサーバー処理を直接呼ぶための仕組み
  - フォーム送信時などにAPIを作る必要があったが、作成しなくてもサーバー側を呼ぶことができる
  - use server付きの関数として実装しておく必要はある
- フォームなどでactionの先に指定することで、データ保存が可能になる
  - 実行はサーバー側で行なわれる
  - use serverの記述が必要
    - 通常サーバー側実行では記述が要らないが、server Actionsで実行される関数では記述が必要

完成したNext.jsを使う

- Next.jsを利用したシステムが完成したら以下のコマンドでBuildをおこなう

```
npm run build
```

- Buildすることで、.nextフォルダー内に成果物が出力される
  - 本番サーバーに格納して、以下のコマンドで実行する

```
npm start
```

- SSRか、SSGなどで出力されるかどうかはNext.jsが自動的に判別して出力する
  - SSGは意識して作らないとダメなので、ほとんどはSSRとして出てくる
  - 強制的に出力形式を指定することも可能ではあるが、要件を満たしていないとエラーになる

以降の学習

- とほほのNext.js入門
  https://www.tohoho-web.com/ex/nextjs.html
  - 少し古いかもだけど、読みやすいはず
- Next.jsの考え方
  https://zenn.dev/akfm/books/nextjs-basic-principle
  - いきなり読むと難しいかも、、
- Learn Next.js
  https://nextjs.org/learn
  - 公式のチュートリアル。英語だが訳せばいけるし、幅広い

ありがとうございました。

Next.jsは奥深く進化が速いですが、
サーバーとクライアントを同時に
作成できるなど、利点が多いです。

良き活用方法を見つけてください