HF ドップラー観測データの利活用を目的とする Web アプリケーションのレンダリング設計

5421 中嶋 柊（指導教員：才田 聡子）

HF Doppler Observation Data Utilization Front-End Design of Web Application

概要： 電離圏研究を目的とする短波ドップラー (HFD) の観測データを保管しているユーザー体験の向上を目的として,Web アプリケーションの新規実装を行うものである.
本研究ではアプリケーション実装におけるデータの取得と画面描画の高速化に視点を置き, Next.js を用いたページレンダリングの最適化を目指し実装・検証する.

キーワード Web アプリケーション フロントエンド ページレンダリング TypeScript Next.js

1. はじめに

電離圏研究を目的とする短波ドップラー (HFD) の観測データは, 自然科学分野の研究のためオープンデータとして公開され自由な使用が許可されている.
現状, 当データの効率的な再利用を推進するためデータのグラフィカルな表示を行う数件の web アプリケーションが開発されているが, その多くが開発運用やユーザーエクスペリエンス (以下 UX) 等の観点での問題点を抱えている.
本研究では既存のアプリケーションからの新規アプリケーション作成に伴うデータ取得とページのルーティングを通した実装および検証を行う.

2. 使用技術

2.1 Next.js について

Next.js とは Vercel 社が提供する Web フレームワークである. JavaScript の UI ライブラリの一つである React.js がベースとなっており, その最大の特徴の一つが多様なページレンダリング手法のサポートである. このフレームワークは, 開発者がアプリケーションの特定のニーズに応じて最適なレンダリング戦略を選択できるように設計されている.
Next.js のレンダリングオプションには, 一般的なクライアントサイドレンダリング (CSR) の他に, サーバサイドレンダリング（SSR）, スタティックジェネレーション（SG)、およびインクリメンタル静的再生成（ISR）が含まれる.
2024 年現在, バージョン 12 系以前で用いられていたクライアントサイドでのレンダリングをデフォルトとした Page Router とバージョン 13 系で追加されたサーバーサイドでのレンダリングをデフォルトとした App Router の 2 種類の Routing 手法が存在する. 本研究では Page Router を用いることとした.

2.2 ページレンダリング手法

2.2.1 ページレンダリングとは

Web アプリケーションにおいてページレンダリングとは, ウェブサーバーまたはブラウザが HTML/CSS/JavaScript などのコードを解釈し, ユーザーのデバイス上で視覚的なページとして表示するプロセスのことである. プロセス内の各ステップを実行するタイミング等により分類されており, 適切な手法を選定することによりサイトの高速化やユーザ体験の改良, SEO の改善等が期待できる. 次に代表的なレンダリング手法とそのメリット・デメリットについて記す.

2.2.2 SSR

SSR（サーバーサイドレンダリング）はウェブページのレンダリングをサーバー側で行う手法である. この手法ではユーザーのリクエストに応じてサーバーが HTML を生成し, 完成したページがクライアントに送信される.
SSR はリクエストが起きたタイミングでクライアントサイドと比較して高速なサーバーサイドで HTML が生成されるため, 初回ページロードの速度を向上させ SEO 対策に有効である. 他にも完成された HTML が送信されるため SEO に強い点や, クライアントサイドの JavaScript 実行環境に依存しにくい点がメリットである. しかし, サーバー負荷が大きい点やページ間移動にリロードが伴う点, 後述する SG や ISR と比較するとコンテンツの表示に時間が掛かる点等がデメリットとして存在する.

2.2.3 SG

SG（スタティックジェネレーション）は, ビルド時に静的な HTML ファイルを生成しておく手法である. このアプローチではビルド時にすべてのページが静的な HTML として生成され, リクエストに応じてその HTML が提供される.
ビルド時に HTML が生成されレスポンスの際にはそれを提供するだけのため非常に高速であることがメリットとしてあげられるが, ビルド後の変更が困難なため動的なページ生成には向かないというデメリットが存在する.

2.2.4 ISR

ISR（インクリメンタルスタティックリジェネレーション）は, 静的サイトの生成とサーバーサイドレンダリングを組み合わせた手法である. ISR では, ビルド時に一部のページのみを静的に生成し残りのページはユーザーのリクエストに応じて生成されキャッシュされる. これにより, サイトのビルド時間を短縮しつつ動的なコンテンツの提供が可能となる.

静的生成と動的生成のバランスが取れパフォーマンスとコンテンツの更新の柔軟性が両立されているメリットがあるが, 再生成後初回リクエスト時のオーバーヘッドが大きく読み込みに時間がかかってしまう場合があるというデメリットが存在する.

2.2.5 Next.js におけるページレンダリング

Next.js のページレンダリングは主にページコンポーネントのライフサイクルとデータ取得方法に依存する特徴があり, それらのレンダリング手法をページごとに異なる方法で適用できる柔軟性にも優れている.
例として一部のページは SSR を使用して動的にデータを取得を行い、他のページは SG を使用して静的に生成することも可能である. このような柔軟性により開発者はパフォーマンス, SEO, ユーザーエクスペリエンスのバランスを取りつつ, アプリケーションの特定の要件に合わせて最適なレンダリング戦略を選択できる. これにより, 開発者はより高速で対話的なウェブアプリケーションを構築することができ, 最終的にはユーザーに優れた体験を提供することが可能となる.

2.3 スクレイピングについて

このアプリケーションは, 各観測所にて観測されたデータを素のディレクトリ構造のまま公開されている既存のアプリケーションからデータをスクレイピングしてきて再描画を行うものである. スクレイピングとは web アプリケーションから HTML データをクローラによって取得し、必要とされる情報を抽出するする技術のことである.
スクレイピングの際には取得元のサイトの規約に乗っ取る義務がある. 今回の場合は取得先のデータが OSS 化されていることと学術目的として取得先の団体からの許可を得ていることから, 過剰な負荷がかからない範囲内での利用が可能であると判断した.

3. アプリケーション設計と実装方法

3.1 実装の要件

本研究ではアプリケーション実装のうち外部データの取得とページの描画を主体としている. そのため UX 向上を目的とした, データフェッチからの画面描画速度の高速化を目標に設定した.
今回作成するアプリケーションのうち, web スクレイピングを伴うページは大きく分けて以下の 2 つである.

- 波形データの検索ページ
- 波形の描画ページ

これらのページの実装を行い、ビルド時間とスクレイピングのリクエスト数の 2 軸から最適化を行うものとした.

3.2 スクレイピングに用いた関数

今回実装したスクレイピング関数は大きく 2 つあり, 1 つは指定した日時・観測所の周波数の CSV データを JSON 化して取得するもの, もう 1 つは存在するデータの一覧を取得するものである. スクレイピング先のアプリケーションでは観測所・年度・日時の順にネストされたパスから周波数のデータを取得可能である. 1 つめの関数では URL で具体的なデータの所在ページを指定した上でスクレイピングした. 2 つめの関数では存在するページの一覧化のため観測所のひとつ上の階層から子要素となるページを再帰的に全探索する形でスクレイピングし整形を行った.

3.3 レンダリング設計

次にレンダリング設計について述べる. 以下の 2 パターンの設計で実装を行い、ビルドにかかった時間を比較し考察することとした.

1. ISR を用いる方法
2. SG と SSR を用いる方法

3.3.1 ISR を用いる方法

1 つ目の手法は ISR を用いた実装である. この手法はビルドの際に一括でスクレイピング処理を完結させて静的なサイトを生成するものである.
実装方法は getStaticPaths と呼ばれる関数内でディレクトリ一覧をスクレイピングし静的にビルドするページを確定させた上で, getStaticProps という関数でそれら一つ一つのページの波形データを取得してビルドを行う.
また ISR 技術を用いて再ビルドのタイミングを制御することにより, 一定期間ごとに新しく追加されたデータも取得できるようにさせることができるようにする.

3.3.2 SG と SSR を用いる方法

2 つ目の手法は SG と SSR を併用させる方法である. この手法ではディレクトリの取得と波形データの取得をそれぞれ SG・SSR をもちいて分離させる方法である.
実装方法は getStaticProps を用いて検索ページに存在するパスを静的に生成し, getServerSideProps を用いて各ページでリクエストされた際に動的に波形データを返すというものである.

3.4 実行環境

今回の実行環境とした PC を表　に示す.
また使用したパッケージのバージョンを表　に示す.

表 1　実行環境

名称	MacBookPro
CPU	Apple M1 Max
メモリ	32GB
コマンドシェル	zsh (v1.85.1)

表 2　パッケージのバージョン

名称	バージョン
Node.js	18.12.0
Next.js	13.0.6
TypeScript	4.9.4

4. 実装結果

4.1 スクレイピング関数の実行時間

実装した 2 つの関数の実行時間を示す. なお実行時間計測には performance.now() メソッドを用い, 関数の実

行前後の時間差から計測している.

4.1.1 scrapeWaveData

この関数は, 引数で渡したデータ形式, 観測所名, 年, 日時から観測データの CSV データを取得し JSON 形式に変換するものである. 実行時間を表 に示す

表 3 scrapeWaveData の実行時間

回数	実行時間 (ms)
1 回目	1435
2 回目	1166
3 回目	1054
4 回目	997
平均	1163

結果としておよそ 1 秒でデータの取得ができることが確認できた.

4.1.2 scrapeDirectory

この関数は, 引数で渡した URL 配下に存在するページを再帰的に全探索し, ディレクトリ名をキーとするネストされたオブジェクトとして返すものである. 実行時間を表 に示す.

表 4 scrapeDirectory の実行時間

回数	実行時間 (ms)
1 回目	319411
2 回目	325276
3 回目	263359
4 回目	261885
平均	292482.75

結果として実行には 300 秒前後かかることが確認できた.

4.2 静的ビルドの実行時間の比較

ISR を用いる手法と SG と SSR を併用する方法との 2 パターンでビルドを実行し, ビルドログに出た実行時間を示す.

4.2.1 ISR を用いる方法

ISR(SG) を用いて, ビルド時に存在するパスを生成した後各ページまでを静的に生成する方法を用いた場合のビルドログを図 1 に示す.
実行結果からビルド時間は 1847 秒であることが確認できる.

4.2.2 SG と SSR を用いる方法

SG と SSR を併用して, ビルド時には存在するパスのみを静的に生成しておき各ページはアクセスされた際に動的に生成した場合のビルドログを図 2 に示す.
実行結果からビルド時間は 1847 秒であることが確認できる.

5. 考察

5.1 スクレイピングの実行結果について

スクレイピングの実行結果より, ディレクトリ情報取得は波形データの取得と比較して時間がかかることが確認できる. これは波形データの取得は一度のページアクセスで済むのに対して, ディレクトリ情報の取得は

```
See more info here: https://nextjs.org/docs/messages/large-page-data
Warning: data for page "/data/[type]/[loc]/[year]/[date]" (path "/data/D
GL/SGD/2024/0108") is 1.54 MB which exceeds the threshold of 128 kB, thi
s amount of data can reduce performance.
See more info here: https://nextjs.org/docs/messages/large-page-data
info  - Generating static pages (9051/9051)
info  - Finalizing page optimization

Route (pages)                                       Size     First Load
JS
┌ ○ /                                               298 B          73.4
kB
├   /_app                                           0 B            73.1
kB
├ ○ /404                                            193 B          73.3
kB
├ λ /api/hello                                      0 B            73.1
kB
├ λ /api/pagelist                                   0 B            73.1
kB
├ ● /data/[type]/[loc]/[year]/[date] (13981927 ms)  387 B          73.5
kB
├   ├ /data/DGL/CHB/2020/1206 (3194 ms)
├   ├ /data/DGL/CHB/2020/1208 (3171 ms)
├   ├ /data/DGL/CHB/2020/1207 (3169 ms)
├   ├ /data/DGL/CHB/2020/1205 (3163 ms)
├   ├ /data/DGL/CHB/2020/1209 (3141 ms)
├   ├ /data/DGL/ONA/2021/0411 (2942 ms)
├   ├ /data/DGL/CHB/2020/1203 (2879 ms)
├   └ [+9040 more paths] (avg 1544 ms)
└ λ /ssr                                            2.36 kB        75.5
kB
+ First Load JS shared by all                       75.1 kB
  ├ chunks/framework-4556c45dd113b893.js            45.2 kB
  ├ chunks/main-7922beff118b306c.js                 26.7 kB
  ├ chunks/pages/_app-f84f72e1bc8adae4.js           408 B
  ├ chunks/webpack-69bfa6990bb9e155.js              769 B
  └ css/c90f27a16a174905.css                        1.92 kB

λ  (Server)  server-side renders at runtime (uses getInitialProps or get
ServerSideProps)
○  (Static)  automatically rendered as static HTML (uses no initial prop
s)
●  (SSG)     automatically generated as static HTML + JSON (uses getStat
icProps)

✨  Done in 1847.41s.
```

図 1 ISR(SG) を用いた際のビルドログ

```
Route (pages)                                 Size     First Load JS
┌ ● / (178550 ms)                             383 B          73.5 kB
├   /_app                                     0 B            73.1 kB
├ ○ /404                                      193 B          73.3 kB
├ λ /api/hello                                0 B            73.1 kB
├ λ /api/scrapedir                            0 B            73.1 kB
├ λ /data/[type]/[loc]/[year]/[date]          386 B          73.5 kB
└ λ /ssr                                      2.36 kB        75.5 kB
+ First Load JS shared by all                 75.1 kB
  ├ chunks/framework-4556c45dd113b893.js      45.2 kB
  ├ chunks/main-7922beff118b306c.js           26.7 kB
  ├ chunks/pages/_app-f84f72e1bc8adae4.js     408 B
  ├ chunks/webpack-69bfa6990bb9e155.js        769 B
  └ css/f66332459d380bab.css                  1.91 kB

λ  (Server)  server-side renders at runtime (uses getInitialProps or get
ServerSideProps)
○  (Static)  automatically rendered as static HTML (uses no initial prop
s)
●  (SSG)     automatically generated as static HTML + JSON (uses getStat
icProps)

✨  Done in 187.85s.
```

図 2 SG と SSR を併用した場合のビルドログ

ディレクトリの全探索であり複数回のページアクセスの必要が出てくるためであると考えられる.

5.2 ビルドの実行結果について

ビルド時間を比較した際に ISR を用いた場合は SG と SSR を併用した場合に対して約 10 倍かかってしまうことが見て取れる.

5.3 ビルド方法の選定

これらの結果を踏まえて当アプリケーションでのビルド方法を選定していく.
ISR 生成時間に約 30 分かかってしまうことが確認できた. ISR の仕様として設定した定期時間後の初回アクセス時にビルドが実行されその実行時間がオーバーヘッドとして存在するため, 初回アクセス者に 30 分間の待機時間がかかってしまう. これはユーザー体験面において適当であるとは言えない.
対して SG と SSR を用いた場合にはデータのページに移行する際に約 1 秒のスクレイピング時間がかかる. ページ描画には事前に生成されている静的なデータを返す ISR と比較して時間がかかってしまうがオーバーヘッドの問題を踏まえれば許容範囲と言えるのではないかと考える. SG のデメリットとして動的なページ生成に弱い点が挙げられる. しかし今回のアプリケーションではスクレイピング元のサイトでデータが追加されるのが一日に数回であることから再取得 (再ビルド) すべきタイミングは一日に数回程度であると想定される. この場合は GitHub Actions 等の CI/CD ツールを用いてホスティング先での規定時間ごとの再ビルドを行うことで対応可能であると考察する.
CI/CD ツールを用いた再ビルドは ISR を用いる方法を SG 単体を用いる方法と置き換えることでも実現可能ではあるが, GitHub Actions の料金がジョブの実行時間に依存しているためビルドに過剰な時間がかかってしまう方法では適切とは言えないであろう.
これらの理由をもとに本研究では SG と SSR を併用させて実装することが現状において最適な手法であると結論づけた.

5.4 結果を踏まえての追加仮説

本実験の結果より SG と SSR を併用する方法が適切であると結論づけた. しかしこの選択をした場合にデータページへアクセスした際に SSR で実行されるデータフェッチに約 1 秒かかってしまう点が UX 上の課題点として挙げられる.
これに対してプリフェッチやを活用することでより高速化が望めるのではないかと再仮説を立てた.
プリフェッチとは次にアクセスすると考えられるページのデータを事前に取得しておく技術のことを指す. Next.js には NextLink という機能があり, 現在表示されているページ上にある遷移リンクの情報をプリフェッチしておくことが可能である. 当アプリケーションでの主要なユースケースとして次のフローが考えられる.

1. データの検索ページにアクセスする.
2. データを検索してリンクから任意のデータページにアクセスする.

この仮説はフロー上でデータの検索ページにアクセスした後任意のデータページへ遷移するリンクを表示した時点でプリフェッチを行っておくことで, 実際にアクセスされた後にデータを取得する以上の高速化が望めるのではないかというものである. Next.js のドキュメントや有志による技術ブログを調査をした所によると SSR の際に用いられる getServerSideProps によるトップレベルでのデータ取得に関しては NextLink によるプリフェッチが実行されない様であった. この場合クライアントサイドでのデータ取得を行い次ページのプリフェッチを可能にした方がユーザーの体感上高速化される可能性がある.
また, useSWR と呼ばれるデータ取得のための React Hooks ライブラリを適切に活用することにより, キャッシュ戦略とプリフェッチを最適化できる事も考えられる.
これら仮説に対しては実際に実装してみての検証が必要であり調査する価値は十分あるのではないかと考える.

6. おわりに

本研究では電離圏研究を目的とする短波ドップラーの観測データをスクレイピングして再描画するアプリケーションでのレンダリング戦略に関して, 調査と検証を行った.
この研究結果及び実装に適切な UI 実装を行うことで, 高速で UX 的観点上優れたアプリケーションの作成が可能であると考える.
また考察時に述べた追加仮説に関しても調査の上導入することにより UX 向上の観点で有益な実装となるのではないだろうか.