大規模メディアでのHono実運用

「現代ビジネス」でのNext.jsからの移行

2025 / 10 / 18

KODANSHAtech

矢口裕也

フルスタックエンジニア

経歴
グリー、メルカリ、Indeedを経て
現在KODANSHAtechおよびフリーランス

現代ビジネスとは

現代社会を生き抜くための ビジネス情報ソムリエ

ジャーナリストや専門家による政治経済の徹底分析から、あらゆるジャンルのトレンド情報まで、ビジネスパーソンに上質な情報を提供するWEBマガジン。ここでしか読めない深層的な情報が高く支持されています。

主な仕様・テックスタック

- 読者向けフロントエンド
  - Multi Page Application
  - 高トラフィック
  - **Next.js → Hono**
- 編集者・作家向けCMS
  - Single Page Application
  - 引き続きNext.js

- Node.js
- Pnpm / monorepo構成
- Prisma / AWS Aurora (MySQL)
- Amazon ECS / CloudFront
- S3
  - アセットのみ
- Cloudinary
  - 画像リサイズ・最適化・CDN

主な仕様・テックスタック

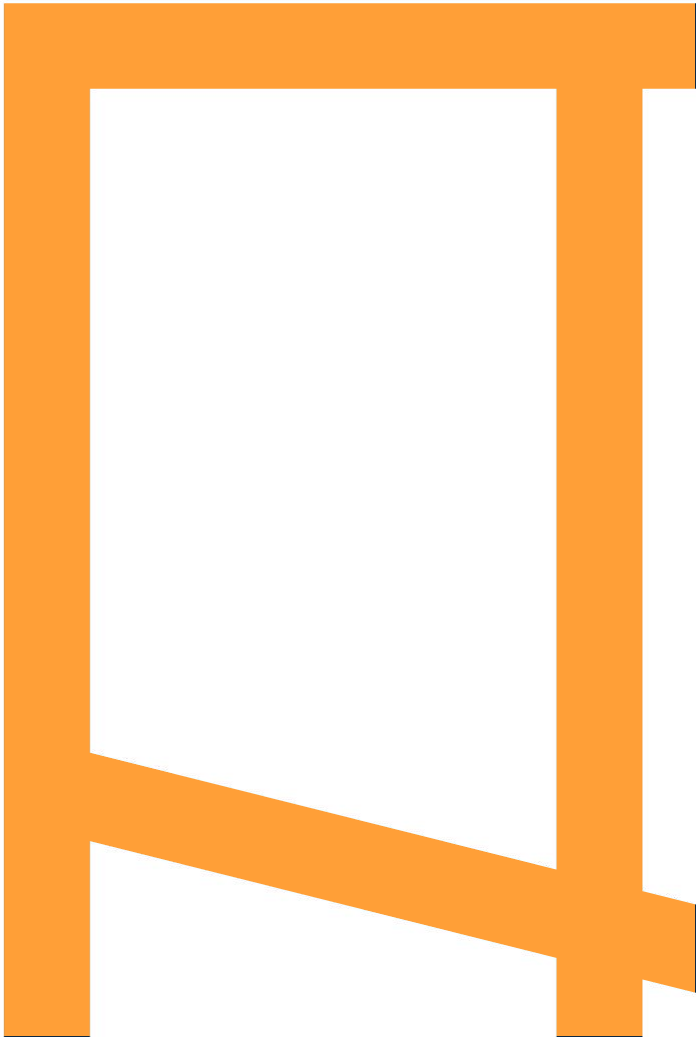
- 読者向けフロントエンド
  - Multi Page Application
  - 高トラフィック
  - **Next.js → Hono**
- 編集者・作家向けCMS
  - Single Page Application
  - 引き続きNext

- Node.js
- Pnpm / monorepo構成
- Prisma / AWS Aurora (MySQL)
- Amazon ECS / CloudFront
- S3
  - アセットのみ
- Cloudinary
  - 画像リサイズ・最適化・CDN

なぜ移行したか

01

課題



Next.js App Routerの課題

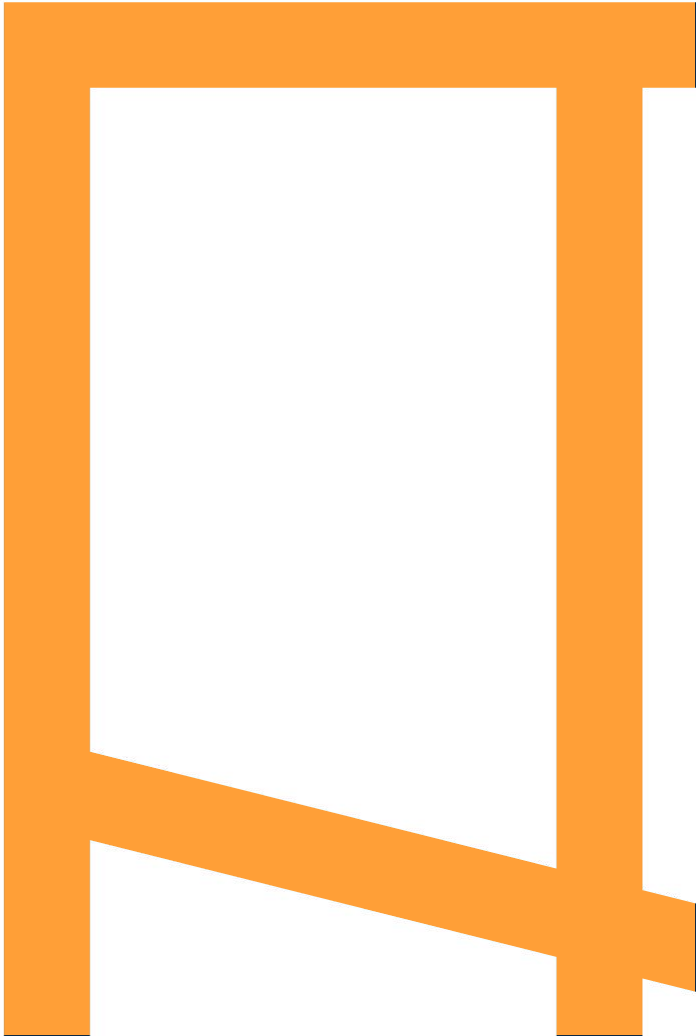
1. レスポンスヘッダーが自由に設定できず、cache-controlヘッダーを最適にできない
2. ページ全体のHydrationのためにブラウザが取得するHTML / JSの容量が大きくなる
3. generateMetadata()がbodyのレスポンスと別れていて、DBから取得したデータなどを共有できない。2度取得するか一時的にキャッシュするなどの対策が必要になる

レスポンスヘッダー設定が自由にできない

- レスポンスをStreaming SSRにすることを強制されているため
- Middlewareという機能があるが、これはルーティングを実施する前にしか使えないため、内容によってヘッダーの内容を分けることができない
- たとえば記事ページで次のような設定をしたいが、これはNext.jsだと**不可能**である
  - 公開前(404)は`max-age`を1分（公開時に見えなくなってしまうため）
  - 公開後は24時間または公開終了時刻の短い方（公開終了時刻のあとに見えては困るため）

02

調査・検討



要件

- Cache-controlを自由に設定できる
  - できないのはNextだけ
- 部分的にHydrationできる
- 工数を増やしすぎない
  - Reactのコード資産を活かせる
  - SSRではasync/await可能にする

代替手段の比較・検証

- Next.jsをパッチ
  - Next.jsが複雑すぎる & Streamingを前提に設計されすぎていて断念
- App RouterをやめてPage Routerにする
- Astro with React
- Honox with React
- Honox with Hono JSX

代替手段の比較

	Cache-control header	部分 Hydration	React or 互換	SSR await w/ Suspense	SSR await
App Router	❌	❌	✅	✅	✅
Page Router	✅	❌	✅	❌	❌
Astro with React	✅	✅	✅	?	❌
Honox with React	✅	自作が必要	✅	修正が必要	❌
Honox with Hono JSX	✅	✅	✅	✅	✅

Reactとasync

Reactでasync componentできるのは次の2つの場合だけ:

- React Server Component
  - SSRに対応しているほぼすべてのフレームワークが非対応
    例外はNext.js App Routerなど一部
- <Suspense>の子要素
  - SSR対応はまちまち、未対応だと中身は実行されない
  - Suspenseが順次解決されるstreamとなり、差分を更新する<script>のチャンクが順次送られる形となる。
    - これはCDNでキャッシュしたい場合、転送量とブラウザの負荷をいたずらに増やすだけになる
    - onAllReady() callbackを利用すれば避けられるがそういった実装はほぼないため、自分で修正が必要

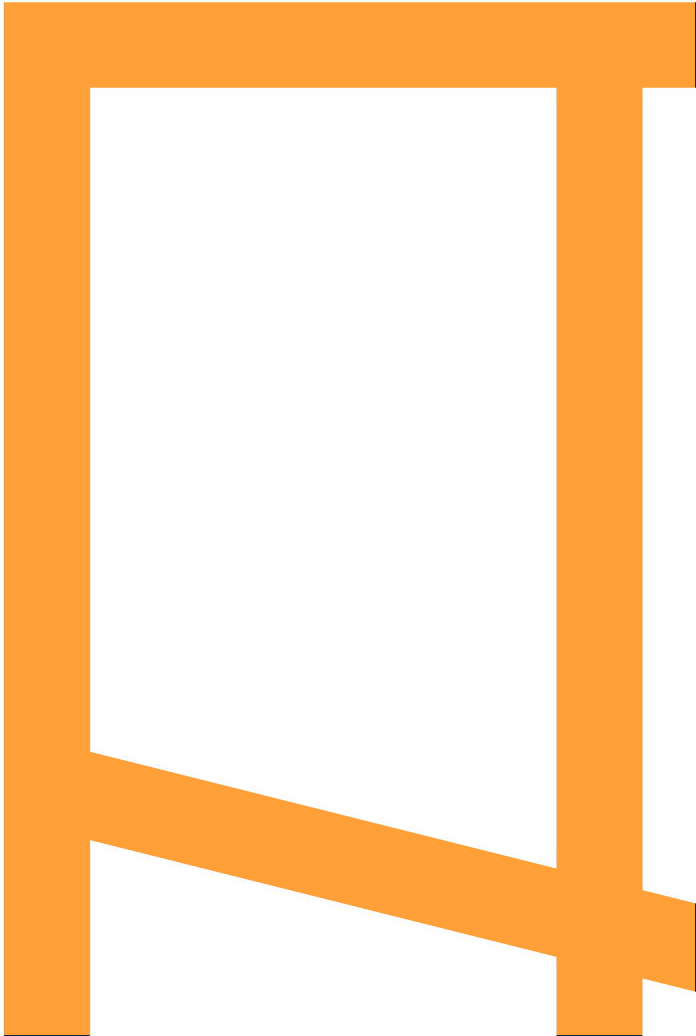
Hono JSX と async

Hono JSXはそういった制約はなく、async componentはPromiseの解決を待ってレンダリングされる

現ビジでは使用していないが <Suspense> を使うことでReactと同じようにstreamで順次レスポンスを返すこともできる

03

フレームワーク整備・開発



Honoxとは

- File-based Routing / SSR / Island Hydration を備えたフルスタックフレームワーク。
- ViteでDevサーバー・production build機能も提供される

→まさに我々の要件に最適！試してみることに

ところがViteが動かない

- 実際の現ビジのコードを追加していくとimportまわりでエラーが大量に出て動作しない
- 直しても直しても様々な箇所でエラーが出る

何がおきていたか (1)

Viteはすべてのmodule loading (サーバーサイド含む）をESMを基本としつつ独自実装している
1. CommonJSが含まれているときの挙動が完璧ではない
2. Monorepoを利用しているときの挙動が完璧ではない

1と2が合わさると手が付けられなくなる。`commonjsOptions` / `optimizeDeps` / `ssr.external` 等のオプションを設定するよう書かれているが、実際に手を動かしたりViteのコードを読んでも対処が難しかった。

何がおきていたか (2)

さらにややこしいことにViteには4つの動作状態がありそれぞれで挙動が大きく異なる

- Dev Server & Build
  - Dev Serverではバンドラにesbuildが使われる
  - Buildではrollupが使われる
  - この2つはコマンド体系なども全く別で、Viteごしに設定を上書きする場合もそれぞれで異なる設定が必要になることも多い
- Client & SSR
  - SSRは後付けなので設定方法や仕様がごちゃごちゃしている

何がおきていたか (3)

```
export default defineConfig(({ command, mode, isSsrBuild }) => {
  if (command === 'serve') {
    // Dev Serverの設定
    if (isSsrBuild) {
      return {}; // SSR 時の設定
    } else {
      return {}; // Client の設定
    }
  } else {
    // command === 'build'
    // build の設定
    if (isSsrBuild) {
      return {}; // SSR 時の設定
    } else {
      return {}; // Client の設定
    }
  }
});
```

この2x2のマトリクスになることは明示されておらず、ドキュメント全体を読んで発見する必要がある

トラブルになり設定を修正する場合はこの4つの状態すべてで正しく動作するように修正しなければならない

またesbuildとrollupの振る舞いにも詳しくなる必要がある

ということで

しばらく格闘した末に
Viteの利用をあきらめました

HonoxでViteが提供していたこと

1. JavaScriptのASTの改変インターフェイス
   Honoxの自動Hydration
2. バンドル機能およびts/tsxファイルのインポート、管理
3. HMR (Hot Module Replacement)
   注: Honoxでは単に再起動をかけるようにしていた
4. 各種プラグイン機能（Tailwindなど）

色々やっていくれていた。これらの代替手段を見つける必要がある

HonoxでViteが提供していたこと

1. JavaScriptのASTの改変インターフェイス
   Honoxの自動Hydration
   → 自動をあきらめて手動でやる
2. バンドル機能およびts/tsxファイルのインポート、管理
   →自分で実装
3. HMR (Hot Module Replacement)
   注: Honoxでは単に再起動をかけるようにしていた
   →tsxの機能でフルリロードをかける
4. 各種プラグイン機能（Tailwindなど）
   →TailwindはCLIから直接利用

Hydrationとは

- SSRでHTML文字列になったコンポーネントをブラウザ側で再度Reactなどの管理化に置き、イベントハンドラの追加やprops変更による動的更新を可能にすること
- 次のような物がクライアント側に必要
  - コンポーネントのJSコード
  - 元々渡されていたprops
  - ハイドレーションするかを決定する目印
    （独自タグ, `id`, `class`, `data-*` propertyなど）
  - Reactなどブラウザで動作するライブラリ

余談: ブラウザで動作するライブラリ？

- Reactなどはブラウザで動作しDOMを操作できるのが当たり前
- Hono JSXは元々SSRでしか動作しなかった。実はDOMを操作できない
  - 代わりにサーバーでVirtual DOMが不要で高速などのメリットもある
- ということで登場したのが `hono/jsx/dom` (Client Components)
  - 元々の `hono/jsx` と完全互換のインターフェイスでブラウザ上で動作する

Honox の Hydration (使い方)

コンポーネントを app/islands に配置するか $componentName.tsx のように $をつけるだけ

```tsx
// app/islands/counter.tsx
import { useState } from 'hono/jsx'

export default function Counter() {
  const [count, setCount] = useState(0)
  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={() => setCount(count + 1)}>Increment</button>
    </div>
  )
}
```

```tsx
// app/routes/index.tsx
import { createRoute } from 'honox/factory'
import Counter from '../islands/counter'

export default createRoute((c) => {
  return c.render(
    <div>
      <h1>Hello</h1>
      <Counter />
    </div>
  )
})
```

Honox の Hydration (実装)

- Vite Pluginとして実装されている
- SSR / Client共にtranspile時に構文木を探してisland化できるところがあったら<HonoXIsland>で囲む。またそこで渡されていたpropsもclientで再取得できるようにエレメントのdata-*に格納。
- ソースコード全体を囲んでいきつつislandコンポーネントのリストを作成。レスポンスに含める
- client向けにこのリストをもとにコンポーネントのコードをbundleする
- ブラウザではリストからコンポーネントを探してhydrationする

手動 Hydration (使い方 1)

コンポーネントは明示的にラップする必要がある。呼び出す方は変更不要

```
export const CounterRaw = () => {
  const [count, setCount] = useState(0)
  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={() => setCount(count + 1)}>Increment</button>
    </div>
  )
}

export const Counter = $island(CounterRaw)
```

手動 Hydration (使い方 2)

client側のコードで明示的にimportしてhydartionリストに追加が必要

```
// client.ts
import { CounterRaw } from './components/Counter';
…

const components = {
  CounterRaw,
  …
};

hydrateAllIslands(components);
```

手動 Hydration (実装 1)

$islandはただのラッパー関数

引数を で囲いつつdata-*にpropsを追加

```tsx
export function $island<P extends object>(
  Component: React.FC<P>,
  className?: string
): any {
  const componentName = Component.displayName || Component.name;
  const IslandComponent: React.FC<P> = (props) => {
    return (
      <div
        data-app-hydrated="false"
        data-app-component={componentName}
        data-app-props={JSON.stringify(props)}
        className={className}
      >
        <Component {...props} />
      </div>
    );
  };
  // Mark `$` to the island component
  IslandComponent.displayName = `$${componentName}`;
  return IslandComponent;
}
```

手動 Hydration (実装 2)

clientではhydrationが必要なelementを探し、createElement() / render() する

```
import { createElement, render } from 'hono/jsx/dom';

export async function hydrateAllIslands(components) {
  const islands = document.querySelectorAll('[data-app-hydrated]');
  islands.forEach((island) => {
    ... // props, Componentの取得
    const newElem = await createElement(Component, props);
    await render(newElem, island);
  });
}
```

HonoxでViteが提供していたこと（再掲）

1. JavaScriptのASTの改変インターフェイス
   Honoxの自動Hydration
   → 自動をあきらめて手動でやる
2. バンドル機能およびts/tsxファイルのインポート、管理
   →自分で実装
3. HMR (Hot Module Replacement)
   注: Honoxでは単に再起動をかけるようにしていた
   →tsxの機能でフルリロードをかける
4. 各種プラグイン機能（Tailwindなど）
   →TailwindはCLIから直接利用

Clientのbundle

ESBuildを直接利用

Devではwatchモード

```
$ esbuild app/client.ts --bundle --outdir=.dev --watch
```

Prodではビルド後にハッシュを付与（例: `client.js` -> `client-12abff33.js`)

```
$ esbuild app/client.ts --bundle --outdir=dist \
  && tsx app/build.ts
```

ビルドしたファイルはコンテナイメージにもアセット用S3にも追加される

サーバー起動時にglobでファイル名を決定。コンテナ内の`dist/client-*.js`を探す

Server

HonoxのFile-based RoutingはViteの`import.meta.glob`に依存している
→通常のglobパッケージで探すようにした。参考実装として公開中
https://github.com/yayugu/hono-file-based-router

実行はtsxで行う

```
$ tsx app/server.ts watch # dev
$ tsx app/server.ts       # prod
```

開発時のCLIと自動リロード

複数コマンドを色分けしつつ同時実行できる concurrently を利用している

```
$ pnpm dev
```

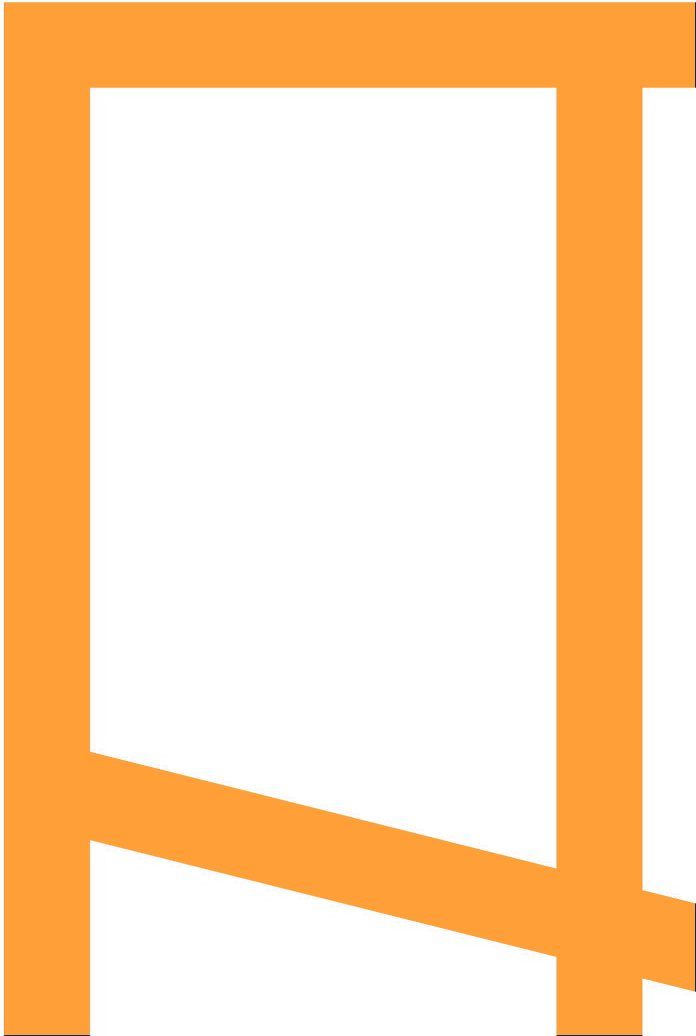
ですべてが起動し、編集すると自動でリロードされる

```
// package.json

"scripts": {
  "dev:server": "tsx watch --clear-screen=false app/server.ts",
  "dev:tailwind": "tailwindcss --input app/style.css --output .dev/style.css
--watch",
  "dev:esbuild": "esbuild app/client.ts --bundle --outdir=.dev --watch",
  "dev": "concurrently -n server,tailwind,esbuild -c green,yellow,blue
\"pnpm run dev:server\" \"pnpm run dev:tailwind\" \"pnpm run dev:esbuild\"",
},
```

```
[server]
[server] > front-hono@ dev:server /home/yayugu/gendai-cms/apps/front-hono
[server] > NODE_ENV=development dotenv -e ../../.env -e ./.env -- tsx watch --clear-scree
n=false app/server.ts
[server]
[tailwind]
[tailwind] > front-hono@ dev:tailwind /home/yayugu/gendai-cms/apps/front-hono
[tailwind] > tailwindcss --input app/style.css --output .duridev/style.css --watch
[tailwind]
[esbuild]
[esbuild] > front-hono@ dev:esbuild /home/yayugu/gendai-cms/apps/front-hono
[esbuild] > esbuild app/client.ts --bundle --outdir=.duridev --watch
[esbuild]
[esbuild] [watch] build finished, watching for changes...
[tailwind] ≈ tailwindcss v4.1.11
[tailwind]
[tailwind] Done in 161ms
[server] Server is running at http://localhost:8787
[tailwind] Done in 2ms
[server] 10:03:07 PM [tsx] change in ./app/routes/index.tsx Restarting...
[server] Server is running at http://localhost:8787
[tailwind] Done in 2ms
[esbuild] [watch] build started (change: "app/client.ts")
[esbuild] [watch] build finished
```

04

移行作業



各ページのルートの移行

- ルートはApp Routerとは非互換なので変更が必要。またApp Routerのmetadataを別で指定する必要のある非効率なして方法もやめたかった
- 1ページを手動で移行
- Coding Agentにそれを真似て同じように作業をさせ指示を出す

```
> migrating from apps/frontend (Next.js) to apps/front-hono (Hono) with chaning the framework.
  Refer articles/-/[articleId]/page.tsx to articles/-/:articleId.tsx migration case and migrate (copy and rewite)
  routes from app/*.
```

- できたものを手動で修正

動的なコンポーネントの移行

- **基本的にはisland化するだけで動作した**
  - React用のライブラリも動作した
    `embla-carousel-react`, `html-react-parser`, `react-hook-form`などなど
  - Hono JSXのバグは何度か踏んだためPRを送り動くようにした（後述）
- JSONにシリアライズできない値を渡していた箇所は
  構造を再検討した（後述）

動的なコンポーネントの移行（Next.js固有）

- next/navigation
  - 次のように置き換え
    - serverでは`c.req/res`
    - clientでは`window.location`, `window.history`
  - 振り返るとclient/serverを透過的に扱えるnext/navigation相当のものを作ってもよかった

動的なコンポーネントの移行（広告）

- 広告（のSDK呼び出し）はほぼフルスクラッチ
  - 元々ReactでuseEffectを大量にしていて非常に読みづらくなっていた
  - SSRしないほうがシンプルになる
  - React互換もやめてVanila JSで直した
    - Hydrationしていない部分であればブラウザでどういじっても問題なくなって楽になった

余談: Hydrationでできないこと

- island境界ではJSONにできるものしか渡せない。関数やDateなどは非対応
- 子要素を渡すこともできない
- (Node（hook / callbackなし）はserializeできるようになってほしい)

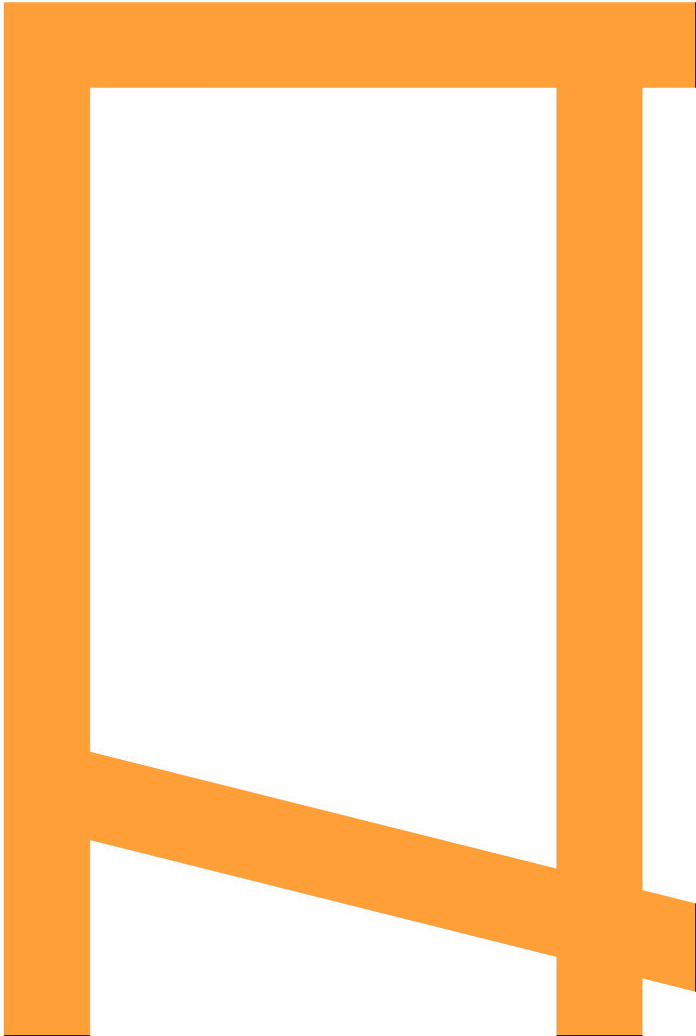
```
// NG: 関数
<IslandComponentA someFunction={() => console.log('Hello')} />

// NG: 子要素
<IslandComponentA>
  <div>Hello World</div>
</IslandComponentA>
```

余談: RSCとの違い

- RSCは内部的にRPCを作っていて開発者には透過的に見せている。そのためclient / server componentを入れ子にできる
  - HydrationではSSR後はserverでの実行はできない。別途RPC / APIをつくって呼び出しが必要
- RSCは大抵のものをserializeして渡せる。function, Date, React Nodeなど
  - Hono JSXではJSONにできるものしか渡せない
  - このためisland境界で子要素を渡すこともできない
  - Hono JSX Nodeはserializeできるようになると嬉しい
- RSCはこれら利便性の代償として恐ろしく複雑でClient / Server / トランスパイラ（Webpack, SWC）を束ねてうまく動作させるように作る必要がある
  - 単なるHydrationは数十行で実現可能

05

移行結果



成果

- サーバー台数が**60%**減
  - 適切な設定が可能になりCDNでより有効にキャッシュできるように
- CDNのデータ転送量（画像除く）が**70%**減
  - Hydrationが部分的になったことでJSやHydration用propsの転送が大幅削減

予想外のメリット

- Devサーバーの起動やリロードが非常に高速になった
  - Next.jsが遅すぎたともいう
  - HMRを使わず毎回プロセス再起動しても十分に高速
- Hydration Errorが減った
  - Nextでは本来Hydration不要なコンポーネントでもHydrationしていた
  - 環境細かい差異を吸収したりエラーを抑制するために無駄にuseEffectなどをしている箇所もあった
  - 開発コストが下がり、不要なコードも減らせた

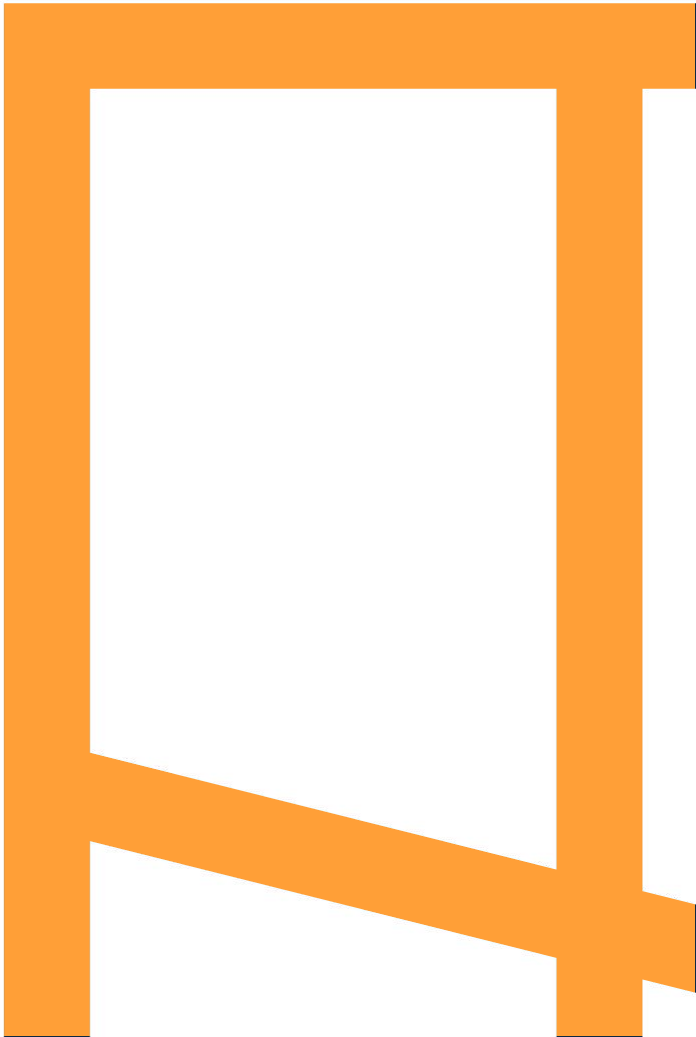
Honoへのcontribution

フレームワークを整備したり、Reactで書かれたコードを動かそうとすることでいくつかのバグを発見・報告できた。いくつかはPRも作成/mergeされている
エッジケースはまだまだ多いが、遭遇してもコードがコンパクトで読みやすくとても簡単に修正できるのはHonoの大きなメリット

- Issue #4235: onError handler couldn't handle Hono JSX errors. Also no error messages on console
- PR #4236: Fix the JSXNode validation. Allow the function type for props.ref
- PR #4257: JSX cloneElement didn't copy children
- Issue / PR #4295: Context.redirect doesn't accept a URL which contains multibyte characters

06

まとめ



HonoはMPAのベストソリューション

- Honoの利用はどちらかというとAPIに寄ったものが多かった
  - Expressの後継としての期待
- 今回の開発と実運用でMulti Page Applicationのためのフレームワークとしての有用性を示せた
- より多くの人がHonoとHono JSXを利用するようになると嬉しい
  - hono/jsx/dom も、もっとつかわれてほしい！！！

ありがとうございました

Thank You

質問・感想・相談など、この後どしどし話しかけてくださいね！