Ameba

リリースから5年、Webフロントエンドの経年劣化と向き合う

2022年10月11日 Muddy Web #3

Ameba

AmebaNewsのシステム概要

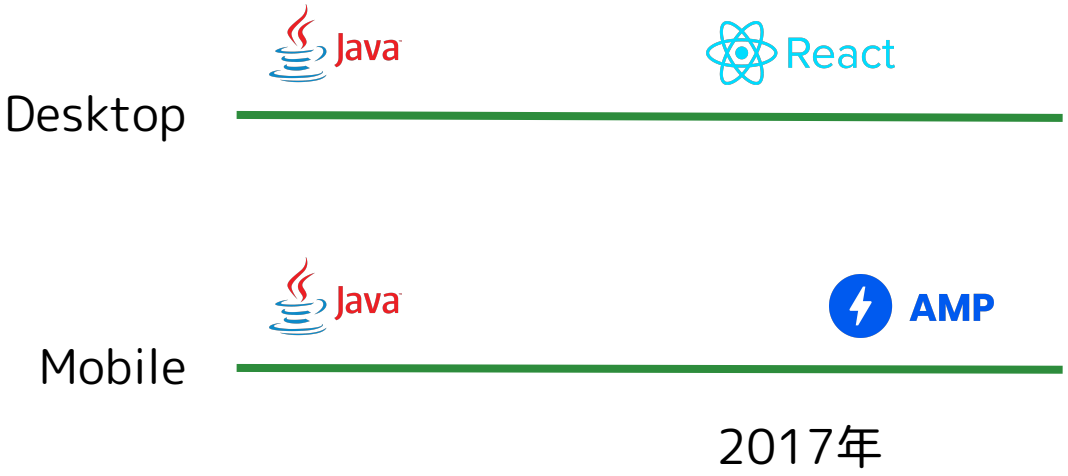
原 一成

AmebaNewsとは

Ameba News

AmebaNewsでは「ちょっと新しい日常を」をコンセプトに、芸能人・有名人のエンタメニュースを中心にお届けしています。中でもAmebaブログ(アメブロ)発の記事や特集などを豊富にそろえて画像や動画と併せて提供しています。

AmebaNews Webシステム概要



Desktop

- MPAとSPAが合わさったいわばIsomorphic Web App
- 大部分のアーキテクチャはAmebaブログを流用
  - React, React Router, Redux

→　悪くはないけどクライアント処理がほぼないAmebaNewsの構成としてはやや過剰…

→　実装時の考慮も多く、移行もしにくい

https://developers.cyberagent.co.jp/blog/archives/636/

Mobile

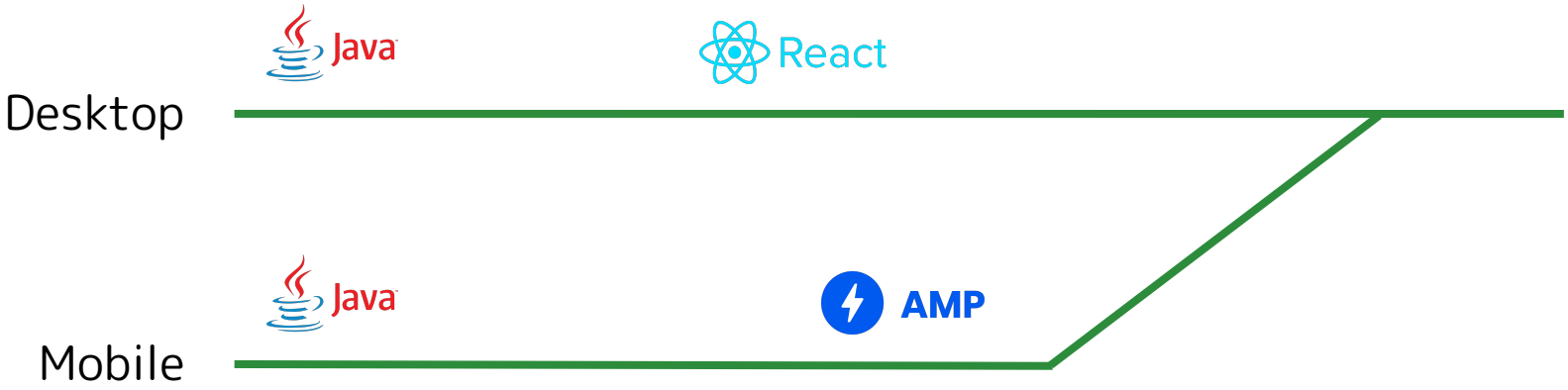
- MPA + CDN (Fastly)
- AMPをフレームワークとして駆使
- 一応レスポンシブ対応済み

→ AMPの利点もなくなってきた

→ 実装時にDesktopとMobileの構成が違いすぎる

https://developers.cyberagent.co.jp/blog/archives/16818/

AmebaNews Webシステム概要



AmebaNewsシステム変更方針

- DesktopとMobileの(Ameba的に)良いところを保ったまま統合する
  - React, MPA + CDN
- 刷新(作り直し)は今回は除外する
  - サービス規模的に少し整えるだけで目的が達成できそう
  - システム構成が軽くなるので、次の移行の選択肢も増えるはず
- 同時に開発環境(テスト含め)の改善も行う
  - 今後の機能追加や次の(プチ)刷新でも役立つように

システム的2021年やりたいリスト #268

Edit New issue

Open hara-kazunari opened this issue on Jul 7, 2021 · 3 comments

hara-kazunari commented on Jul 7, 2021 • edited

2021年は今後開発やメンテナンスがスムーズに行えるようにシステム環境を整えます。新規にシステムを作る楽もありますが、リポジトリ自体大きくないかつ開発メンバーも揃っているので、徐々にアップデートしていく方でチャレンジしてみようと思います。

随時追加していきます。

- [x] レスポンシブ
- [x] AMP配信の終了
- [x] TS化
  - [x] TS開発環境
- [x] テスト追加
  - [x] 単体テスト環境
  - [x] UIテスト環境
  - [x] VRT
- [x] Storybook
- [x] 内製記事CMS完全移行
- [x] yarn化
- [x] 依存モジュールのアップデート
  - [x] Webpack
  - [x] Babel
- [ ] インフラリニューアル (移設)
  - [x] デプロイフロー（Jenkins to Circle CI）
  - [ ] コンテナ化
- [x] RUM
- [x] A/Bテスト
- [x] Linter
  - stylelint
  - Prettier
  - CIで動かした
- [x] MPA
- [x] Reduxやめる
  - Hooksベース

3

Assignees
No one—assign yourself

Labels
None yet

Projects
None yet

Milestone
No milestone

Linked pull requests
Successfully merging a pull request may close this issue.
None yet

Notifications Customize
Unsubscribe
You're receiving notifications because you're watching this repository.

3 participants

Lock conversation
Pin issue
Transfer issue
Delete issue

Ameba®

React with SSRをMPA化する

keiya01

_keiya01

Ameba®

Section1

はじめに

変更前の技術構成

- ReactとexpressをSSRを実現している(Next.jsなどは使用していない)
- ルーティングはreact-router
- storeの管理はRedux
- (5年前でこのスタックはかなり攻めていたのでは、、？)

Ameba®

Section2

どのようにMPA化するか

MPA化の方法

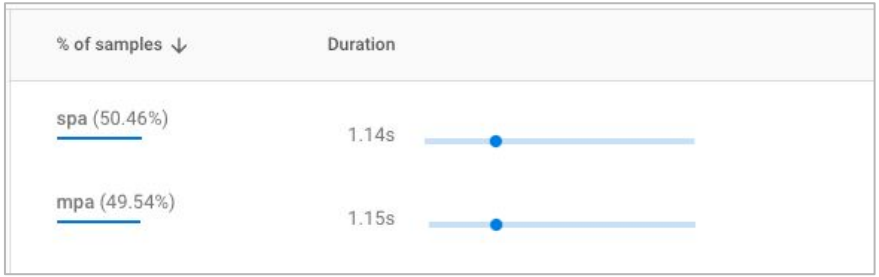
- ページ遷移をMPA的に遷移できるようにすれば良い
- react-routerのLinkをaタグに置き換える
- まずはできるだけ少ない変更で様子を見る

サービスへの影響を確認する

- FastlyでA/B Testing環境を作り、SPAとMPAでサービスへの影響を確認する
  - FastlyのA/B Testingはチュートリアルがあるのでとても簡単でした
  - https://developer.fastly.com/solutions/tutorials/ab-testing/
- PVなどに影響がないことが確認できたら徐々にMPAの影響範囲を広げていく

	MPA	SPA
データ数	92171	91946
平均	1.74922698	1.607922041
中央値	1	1

PPS: SPAが1.6でMPAが1.7



LCP: SPAが1.14s, MPAが1.15s

Ameba

Section3

最適化(リファクタリング)

何をやるか

- MPAになったことで不要なpackagesやシンプルにできる部分があるのでリファクタリングしていく
  - 特にreact-routerとRedux外しについて話します
- テストを通して品質を担保する
- 監視などを通してエラーが起きた時にすぐに対応できるようにしておく
- そもそもエラーの影響を最小限にできるように工夫をする

なぜMPAか

- Newsアプリなので SPA的な遷移は重要ではない
- Clientの依存を減らして初期読み込みを早くしたい
    - メンテナンスの観点でも依存が多いほどメンテナンスが大変
- とはいえReactの書き味を残したい
- 将来的にHydrationを局所的にしたReactみたいなものが出て来ればそれを使えるかも

なぜreact-routerを外すか

react-routerをどうするか考える #681

Closed keiya-sasaki opened this issue on 31 Jan · 5 comments

keiya-sasaki commented on 31 Jan • edited

Specifications

amebanews-webのreact-routerがv3なのでどうにかしたいです。
ドキュメントもまともなものがないので何をするにも不便です。セキュリティ的な懸念もあります。

Development Notes

react-routerをどうするかは以下の選択肢があるかなと思います。

1. react-routerを頑張って最新にあげる
2. 違うライブラリを探してそっちを使う
3. 自前でrouterを作る
4. reactごと剥がして単純なリンクで動く真のMPAにする

4は現実的ではないので1か2または3になるかと思います。

最終的にもはや自前の方が楽という結論(MPAだし)

なぜReduxを外すか

- 昨今の流れとして責務を明確にした上でstateを管理する流れがあるし、
  そっちの方がわかりやすい
- MPAなのでstoreにデータを溜めておくメリットがない
- 今回はグローバルで管理するデータが数えるほどしかないのでstoreは
  React.Contextで管理する
- MPAなのでreact-queryなども使わずfetchをhooksでラップしたものを使い
  特にクライアントでデータのキャッシュはしない

進め方

1. テストや監視周りの準備
2. server側のreact-routerをexpressベースに置き換える
3. server側のredux外し
4. client側のredux外し
5. client側のreact-routerを外す

基本方針

- 進めていくにあたって次の方針を意識しました
- できるだけインターフェイスを変えない
- テンプレートを作ってできるだけロジックを考えずに置き換えができる状態を作る
- 他の施策もあるので既存のコードと新規のコードを共存させつつ徐々に置き換えていく
- 既存の部分を外しやすいように新規のコードを追加していく

テストや監視周りの準備

- initialStateの変更をテストするためにsnapshotテストを行う
  - 意図せずSSRの内容が変わるのを防ぐ
  - puppeteerでHTMLに埋め込まれたstateを取得し、JSONとして保存しておき、変更後に比較する
  - 参考: https://efcl.info/2018/02/02/snapshot-test/
- FastlyのA/B Testingを使い、**Reduxありなし**、**react-routerありなし**のケースを作成し、影響がないことを確認しつつ変更を適用していく
- DatadogとSentryにA/Bのケースのメトリクスを送り、それぞれで変化がないことを確認する

そもそもSSRの仕組み

1. それぞれのページでfetchなどを通してデータを取得
2. Appコンポーネントに取得したデータを渡す
3. Server側でReactのAppコンポーネント(一番親のコンポーネント)をHTMLの文字列に変換する
4. この時にAppコンポーネントではpage(path)に応じたコンポーネントをルーティングする(react-routerなどの役割)
5. HTMLに変換する際にSSRで使用したデータをwindow.INITIAL_STATEなどに入れておいてhydorationできるようにしておく
6. ハイドレーションの際にSSR時のコンポーネントとクライアントが一致するようにクライアントでもルーティングを行う

Server側のreact-routerをexpressベースに置き換える

1. 既存のものとは別に新しく `renderHTML` という共通関数を定義し、この関数にinitialPropsを渡すとSSRされて、SSRしない場合は何も渡さなければOK
2. できるだけ振る舞いを変えないために、react-routerによって定義されていたpropsなどはSSR時に渡すようにする
3. ルーティングのディレクトリ構成はNext.jsに寄せて理解しやすい構成に

Server側のreact-routerをexpressベースに置き換える

- pagesRoutesをエントリーポイントで `app.use(pagesRoutes());` のように呼び出してルーティングする
- `isGoodbyeRedux` の場合、以降の path にはマッチしないので既存のコードに変更を加えることなく新しく書き加えることができる

/routes/index.ts

```ts
const pagesRoutes = () => {
  const app = express();

  app.use((req, _, next) => {
    const abTests = convertABTestHeader(req.headers);
    const isGoodbyeRedux =
      process.env.NODE_ENV === 'development' ||
      abTests[ABTestingMap.goodbyeRedux] === 'true';

    if (!isGoodbyeRedux) {
      return next('router');
    }
    return next();
  });

  app.use('/hoges', hogeIdRoutes());
  app.use('/fuga', fugaRoutes());

  app.get('*', (req, res) =>
    renderHTML(req, res, { status: 404, pageType: PAGETYPE_PC.NOT_FOUND })
  );

  return app;
};

```

Server側Redux外し

1. redux-connectというライブラリを通してSSRしていました
2. Serverのroutingに合わせて新規ページを作り、Redux actionと同じfetch処理を呼び出します
3. fetchしたデータをコンポーネントのpropsに合わせて渡す

Server側Redux外し

- 新規ページのSSR部分
- renderHTMLにinitialPropsを渡す

HogeTemplate.js

```
@asyncConnect([
  {
    promise: async (opts) => {
      const {
        params,
        store: { dispatch },
      } = opts;
      const articleId = params.hogeId;

      return dispatch(fetchHoge(hogeId))
    },
])
@connect((state, ownProps) => {
  const hogeId = ownProps.params.hogeId || '';
  return {
    hogeId,
    hoge: state.hogeReducer[hogeId] || {},
  };
})
export default class ArticleHead extends React.Component {
  render() {
    const { hoge } = this.props;
    return <div>{hoge.value}</div>;
  }
}
```

Before

image

```
// `/:hogeId`の定義
const hogeIdRoutes = () => {
  const app = express.Router({ mergeParams: true });

  app.get('/:hogeId', async (req, res) => {
    const initialProps: InitialProps = {
      status: 200,
    };

    const { hogeId } = req.params;

    const hoge = await fetchHoge(hogeId).catch((e) => {
      console.error(e);
      return null;
    });

    if (!hoge) {
      initialProps.status = 500;
      renderHTML(req, res, initialProps);
      return;
    }

    renderHTML(req, res, {
      ...initialProps,
      hoge,
    });
  });

  return app;
};
```

After

Client側Redux外し

1. useFetchのようなfetch用のHooksを作る
2. ClassコンポーネントをFunctionコンポーネントに書き換える
3. A/Bテストのためにreduxのコードを残しつつ新規のコードを書き足す

Client側Redux外し

- ベースとなるuseFetchの内部でA/B Testing判定をしている
- 各API用のHooksは`useFetch`を経由してAPIにアクセスする

useFetch.ts

```ts
const useFetch = <Data extends unknown>(
  fetch: () => Promise<Data>,
  fetchMore?: FetchMore<Data>
) => {
  const isGoodbyeRedux =
    process.env.NODE_ENV === 'development' ||
    getNewsABTestingValue(ABTestingMap.goodbyeRedux) === 'true';

  if (!isGoodbyeRedux) {
    return undefined;
  }

  // eslint-disable-next-line react-hooks/rules-of-hooks
  return useFetchInner(fetch, fetchMore);
};
```

Client側Redux外し

- 改善前はClassコンポーネントで書かれていた
- Atomic デザイン organisms で fetch していたためどうしても同一コンポーネントに修正を加える必要があった
- Container層など一枚挟んであればもう少し置き換えやすかったかも(教訓)

Hoge.js

```
@connect((state, ownProps) => {
    return {
      hoge: state.hoge;
    };
  },
  { fetchHoge: fetchHogeAction },
)
class Hoge extends React.Component {
  componentDidMount() {
    this.props.fetchHoge();
  }

  render() {
    return <div>{this.props.hoge.value}</div>;
  }
}
```

Before

Hoge.js

```
const HogePresenter = (props) => {
  // `isGoodbyeRedux`の場合はなんらかの値が入ってくる
  const result = useHogeFetcher();

  useEffect(() => {
    // resultが存在する場合は不要にfetchしない
    if (result) {
      return;
    }
    props.fetchHoge();
  }, []);

  const hogeData = result || props.hoge;

  return <div>{hogeData.value}</div>;
}

export const Hoge = connect((state, ownProps) => {
    return {
      hoge: state.hoge;
    };
  },
  { fetchHoge: fetchHogeAction },
)(HogePresenter);
```

After

Client側react-router外し

1. react-router@v3を最新に上げるの辛い
2. universal-router likeなシンプルな独自routerを作る
3. SSR時とHydration時にルーティングを走らせる
4. これを各ページごとに適用していく

Client側react-router外し

- universal-router likeなrouterの定義
- マッチしたコンポーネントを返す
- Code Splittingは未対応

routes.tsx

```tsx
const universalRoutes: Routes = [
  {
    pathname: '/',
    component: (props) => <Home {...props} />,
  },
  {
    pathname: '/hoge',
    children: [
      { pathname: '/:hogeId', component: (props) => <Hoge {...props} /> },
    ],
  },
];

export const findComponentFromRoutes = (
  pathname: string,
  routes = universalRoutes
) => {
  const route = findRoute(pathname, routes);

  return route?.component;
};

```

Client側react-router外し

- Client側でのrouterの出しわけ
- initialPropsにreact-routerを使うかどうかを入れている
  - 使わない場合はlocation系のデータをinitialPropsに入れる
  - initialPropsはSSRで渡ってくるデータ
- Serverの場合は`renderToString`にした上で同じようなコードになる

client.js

```
if (initialProps.withoutReactRouter) {
  // 置き換え後の定義
  const component = findComponentFromRoutes(initialProps.location.pathname);

  render(
    <Provider key="provider" store={store}>
      <ApplicationPage {...initialProps}>
        {component?.(initialProps)}
      </ApplicationPage>
    </Provider>,
    rootElement
  );
} else {
  // react-router用の定義
  render(
    <Provider key="provider" store={store}>
      <Router
        createElement={(Component, props) => (
          <Component {...props} {...initialProps} />
        )}
        history={browserHistory}
        render={
          isGoodbyeRedux
            ? undefined
            : (props) => <ReduxAsyncConnect {...props} />
        }
        routes={routes}
      />
    </Provider>,
    rootElement
  );
}
```

最後にA/Bテストのコードを外す

1. 1ヶ月ほど様子を見たのちに、一気に外していきました。
2. バンドルサイズが合計で **59.66KB(gzip)** 減りました
   - react-router: 33.94KB(gzip)
   - Redux: 25.73KB(gzip)

Ameba®

Section4

まとめ

まとめ

- react-router v3 のドキュメントがないのは辛かったです
- React の SSR 周りの挙動を再理解する良い機会でした
- Router周りも深く知ることができてよかったです

Ameba®

Developer Experience の改善

 高見 駿介

shunke07

Ameba®

はじめに

話すこと/話さないこと

アプリケーションの品質とフロントエンドの責務

- ユーザー体験：パフォーマンス、アクセシビリティ、UI/UX , etc.
- 開発体験：プロダクティビティ、コードの品質 , etc.

→　時流によって変化し、何もしなければ劣化していく…

アプリケーションの品質とフロントエンドの責務

- ユーザー体験：パフォーマンス、アクセシビリティ、UI/UX , etc.
- 開発体験：プロダクティビティ、コードの品質 , etc. **←こちらのお話**

参考

：https://developerexperience.io/practices/good-developer-experience#what-is-a-good-developer-experience

開発体験が良くない：AmebaNewsの場合

メンテナンスコストが高い

- 実装における心理的安全性が低い（実装のハードルが高い）
  - 設計および依存関係がレガシー
  - 静的型付けがない
- コードレビューの負荷が大きい
  - テストコードがない
  - CIでのチェックがなされていない
- ビルドとデプロイに時間がかかる

Ameba®

課題その1

実装における心理的安全性が低い

（実装のハードルが高い）

改善：実装における心理的安全性が低い

- 開発ガイドラインの作成
  - UIコンポーネント、テスト、パフォーマンスなどに関して
- Linterの運用改善
  - husky, lint-staged, stylelintの導入
- 脱 Git Submodule
- TypeScript化

TypeScript化：実装方針

- できるだけ厳密に型付けする
  - compilerOptions の strict オプションを true に
  - @typescript-eslint/eslint-plugin の利用
  - ts-migrateは利用しない
- fetch周りなど重要なロジックを優先して対応する
- 新規の開発では必ずTypeScriptで実装する

TypeScript化：fetch周り

APIリクエスト/レスポンスの型定義はOpen APIをもとに生成して、fetch周りをお型付け（お片付け）

https://github.com/drwpow/openapi-typescript

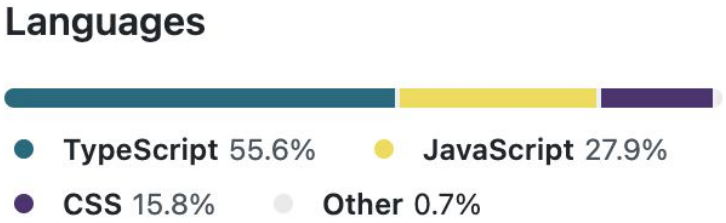
```ts
import type { paths } from '../types/api/generated';

export const requestApi = <
  Path extends keyof paths,
  Method extends keyof paths[Path],
  Query extends paths[Path][Method] extends {
    // eslint-disable-next-line @typescript-eslint/no-explicit-any
    parameters: { query: any };
  }
    ? paths[Path][Method]['parameters']['query']
    : undefined,
  ResponseSchema extends paths[Path][Method] extends {
    responses: {
      [200]: { schema: unknown };
    };
  }
    ? paths[Path][Method]['responses'][200]['schema']
    : undefined
>(
  path: Path,
  query: Query,
  method?: Method
): Promise<ResponseJSON<ResponseSchema>> => {
  // ...
```

TypeScript化：結果

- 15745行を置き換えられた（新規実装分含む）

  → スクリプト全体の約65%
- コンポーネント以外のロジックはすべて置き換え完了



Ameba

課題その2

コードレビューの負荷が大きい

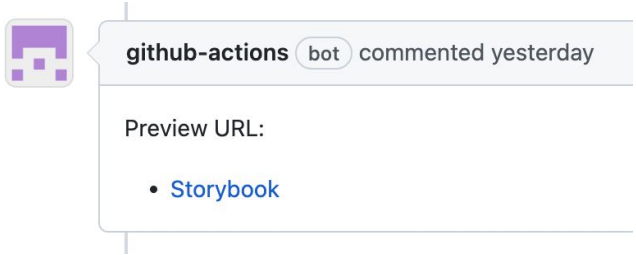
改善：コードレビューの負荷が大きい

- 単体テスト・UIテスト
  - Jest, Mocha, React Testing Library
- CIでのチェック
  - Lint
  - テスト
  - パフォーマンスバジェット：https://github.com/ai/size-limit
- Storybook
  - PRごとのPreview

PR ごとの Storybook Preview

PRごとにStorybookの成果物をデプロイするCIワークフローを作成

- 生成したPreview URLをコメントで通知
- ローカルでのビルドを必要とせずにStorybookの実装を確認できる形に



Ameba®

課題その3

ビルドとデプロイに時間がかかる

改善：ビルドとデプロイに時間がかかる

- Webpack のバージョン更新
  - v3 → v4
- CI の移行
  - Jenkins → CircleCI → GitHub Actions
  - CI上のNodeバージョンを上げる：v8 → v14

Webpackのバージョン更新 (v3 to v4)

- そもそもv3ではStorybookが動作しない
- 段階的にv3からv4への移行を目指す
- 基本は公式のガイドに従う：https://webpack.js.org/migrate/4/

Webpackのバージョン更新：結果

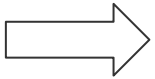
modeオプション(dev or prd)の指定による最適化

https://webpack.js.org/configuration/mode/

→ productionの場合で、平均ビルド時間が**約10秒**ほど改善

Before

```
Version: webpack 3.12.0
Time: 35213ms
```



After

```
Version: webpack 4.46.0
Time: 25018ms
```

CIの移行 (Jenkins to GitHub Actions)

- Nodeのバージョンを上げた (v8→v14) & モジュールをCacheするようにしたことによりビルドなどの実行速度が改善
  - アプリケーションのビルドが**約1分半**ほど短縮
- コード (yml) ベースで各ワークフローを管理できるように

Ameba®

おわりに

まとめ

まとめ

- レガシーなアプリケーションの開発体験を様々なアプローチで改善することができた
  - サービスの安定リリース、リードタイムの改善に寄与することができた
  - 様々なエコシステムに対する理解を深めることができた
- 何もしなければ劣化していく、計画的なメンテナンスが必要
- 刷新はアプリケーションの要件、プロダクトの性質に応じて柔軟に

Ameba

ご静聴ありがとうございました