目次

Webフロントエンド パフォーマンス改善ハンドブック

このパフォーマンス改善ハンドブックでは、ウェブアプリケーションにおけるフロントエンドのパフォーマンス改善について扱っています。

ダウンロード版

埋め込み動画を再生できないなど一部制限がありますが、ダウンロード版を配布しています。

- PDF版
- EPUB版
- MOBI版

目的

このハンドブックでは過去に行った改善の事例を中心に紹介しています。 そのため、現在の最適な解決方法を提案するものではありません。

また、アプリケーションによっても最適な解決方法は異なります。 今回の事例ではViewライブラリにReactを使い映像再生プレイヤーなどある程度複雑な機能を持ったウェブアプリケーションのフロントを扱います。

具体的にはニコニコ生放送（以下「生放送」）で行った事例を中心に書かれています。 開発と平行して行われていたため、React 15から16の間に書かれたものが混在しています。

このハンドブックではどのようにして問題を発見して、何をすれば解決するのかを見つけるという点を重視しています。 紹介している解決方法はただの一例でしかないため、別の解決方法も存在します。 たとえば、今回の事例ではできるだけ元の機能を維持して修正していますが、デザインを変更して根本的に変えてしまうことも正しいはずです。

機能とパフォーマンスに相関がある場合もあります。その場合、そこにはトレードオフの関係があります。 そのようなトレードオフについての考え方の一例として軽量化のトレードオフの検証などもハンドブックに含まれています。

読み方

このハンドブックでは大きく分けて次の2種類について扱っています。

- 視聴中のパフォーマンス改善
- ページロードのパフォーマンス改善

視聴中のパフォーマンス改善

視聴中のパフォーマンス改善はランタイムにおけるパフォーマンス改善について書いたものです。 ランタイムとはページ上のUIの反応速度や画面の更新などウェブページの実行時の動作のことをいいます。

ランタイムのパフォーマンスは操作したときにスムーズに動くかなど操作感に影響を与えます。 また、動画再生やゲームなどはユーザーが何も操作していないも常に描画を更新しています。 そのため、ランタイムのパフォーマンスが安定して良くないと映像がカクカクしたように見えるなどフレームレート（FPS）の低下を引き起こします。

またアプリケーションによってランタイムの性質は大きく異なります。 このハンドブックではReactを使って開発されているニコニコ生放送の視聴ページを例に、ランタイムのパフォーマンス計測や改善について扱っています。

ページロードのパフォーマンス改善

ページロードのパフォーマンス改善はURLにアクセスしてページが表示されるまでのパフォーマンス改善について書かれたものです。 ページロードに関してはすでにさまざまな解説や書籍が存在するので、その中のファイルサイズについてを中心に扱っています。

ウェブアプリケーションを開発していると、動作自体には影響がないためJavaScriptやCSSなどのファイルサイズを見落としがちです。そのようなファイルサイズの問題をどのように見つけるか改善のアプローチについてを扱っています。

視聴中のパフォーマンス改善

パフォーマンス改善や何をするに当たっても目的（方向）と指標がないと、解決しようとしたものごとが本当に解決できたのかわからずに終わってしまう問題があります。

そのため、まず最初に今回のパフォーマンス改善は何を目的にしているのかをはっきりさせることから開始しました。 実際に定めた目的は次のとおりです。

目的

視聴ページを閲覧中（視聴中）に映像がペタペタしたり、コメントがカクカクしたり、ユーザー体験が良くない状態をできるだけ改善/軽減したい。

目的ではないこと

- ページロードの改善
- ファイルサイズの改善

視聴ページの例

> 視聴ページ: 実際に生放送の映像を閲覧するページのこと

目的では、生放送の映像を視聴中の動作を安定化することを目的としました。 このパフォーマンス改善ではランタイムの動作を改善することが目的となりました。 そのため、ページロードの改善（表示速度の改善）はひとまず目的外と設定しました。

また、これに合わせたIssueを管理するためのメタIssueを作りました。 パフォーマンス改善は基本的に総合的なものになるため、さまざまなリポジトリをまたいで改善する必要があります。 これらからクロスリファレンスとなるIssueを作って置くと、後から改善内容をたどるのに役立ちます。

実際に作成したメタIssue:

パフォーマンス調査と改善タスク #674

Closed live opened this issue on 16 Nov 2017 · 16 comments

live commented on 16 Nov 2017 • edited ▾ Member

概要

LeoPlayer、PcWatchPage含め生放送Watchページのパフォーマンス調査し、改善するタスクや調査手法についてを書いていく。

関連あるPRのクロスリファレンス用のIssue。

目的

Watchページを閲覧中(視聴中)に映像がペタペタしたり、コメントがカクカクしたり、ユーザー体験が良くない状態をできるだけ改善/軽減したい。

目的ではないこと

- 起動速度の改善
- ファイルサイズの改善

指標(仮)

視聴中における体験を良くするのが目的なので、指標は明確に定める事が難しい。
視聴中に継続的に取る値として次の項目を取得し、その傾向を参考にする。
環境に左右されやすいRUM(Real User Measurement)なので、強く判断材料にはしにくい(データ量がある程度必要)

また、ボトルネックとなっているところを明確に見つけてそこに関するマイクロな改善を重ねることで最終的な目的を達成する。

指標となるログデータ

#676 で実装されている。

指標

ページロードのパフォーマンスではなく、視聴ページを閲覧中（視聴中）のランタイムのパフォーマンスであるため指標を決めるのが難しい問題があります。

ページロードに関する指標として機械的に取れるものも多く、SpeedCurveやCalibreなどのサービスも対応しています。

- Web クライアントサイドのパフォーマンスメトリクス — Speed Index、Paint Timing、TTI etc... ::ハブろぐ
- Leveraging the Performance Metrics that Most Affect User Experience | Web | Google Developers

これに対して、ページロード後の閲覧中（視聴中）におけるパフォーマンスの指標はアプリケーションに依存するため明確なものがあまりありません。

今回は簡単な指標と計測のツールを作ることにしました。

指標を作る

視聴ページのメインコンテンツは映像、映像の上に描画するコメントです。 そのためどちらかを軸にして、指標とするのが良さそうです。

何度か観測していると、映像が追いついていない時はコメントもカクカクしていることが多いに気づきました。 コメント表示にはCanvasで描画する `comment-renderer` というモジュールを使っています。 そのため、コメントがカクカクしているかは `comment-renderer` からFPSを取得すれば機械的に取得できます。

コメントのFPS

ChromeのCPUスロットリングで6倍遅い状態にしてみると、些細な操作でコメントがカクカクしていることがわかっていました。 コメントがカクカクしてくると映像がとまり（Stalledを引き起こす）ローディングが表示されることも多いことがわかりました。

閲覧中（視聴中）に常に変化のある値の方が指標としては分かりやすいので、コメントのFPSを指標としてみることにしました。 映像がカクカクしているかは技術的に取得することが難しかったという問題もあります（Stalledの回数などは取れるが、それだと0か1になって極端）。

通常60FPSでコメントをレンダリングできますが、処理が重くなってくると30FPSを切って見た目的にもカクカクしてきます。 このFPSの高く保つことができるようになれば、閲覧中がある程度快適に行えていると判断できます。

指標

コメントのFPSを縦軸にし、時間経過でそのFPSがどのように変化しているかを関連付ければ、何が原因（イベント）でFPSが低下しているかもわかりそうです。

パフォーマンスの測定を行うときにはその測定処理自体が重くなることがあるので、 `performance.mark` APIをベースに、コメントのFPSとイベントのデータを計測する処理を実装しました。

具体的な実装方法やライブラリについては次の記事で解説しています。

- performance.markにメタデータを紐付けできるライブラリを書いた | Web Scratch
- `performance.mark` with metadata is useful for Real user monitoring

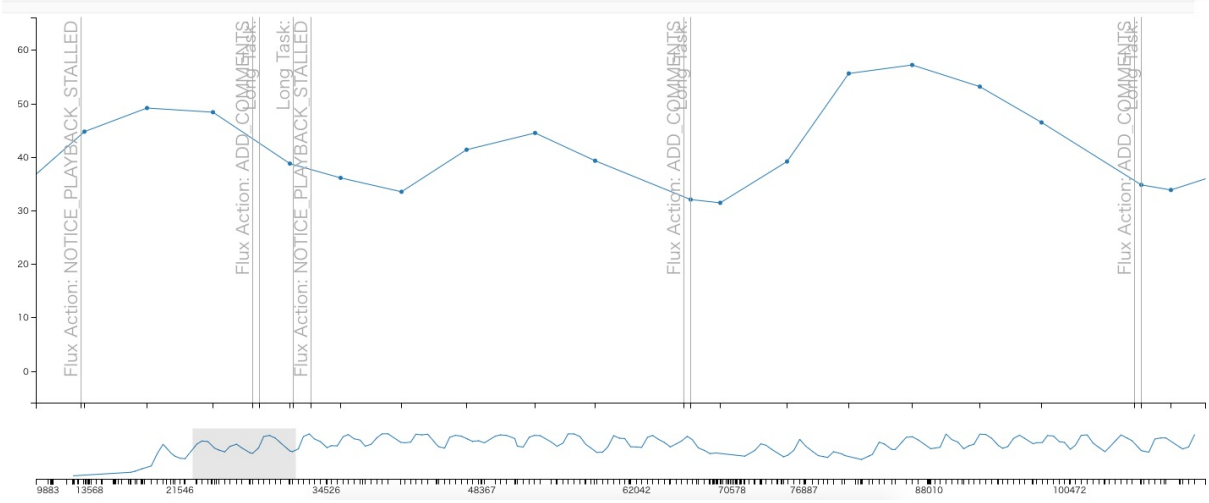
具体的に実装したものは次のとおりです。

- [x] コメント描画のFPS
  - 概要: コメントを描画するレンダラーのFPS（Frame Per Seconds）
- [x] 映像を再生するVideo要素の `VideoPlaybackQuality`
  - 概要: VideoPlaybackQualityは再生時にDropしたフレーム数を取得できる
- [x] 映像の再生詰まりの回数
  - 概要: 映像取得が再生に追いついていないときにローディング表示がでている回数と同義
- [x] Long Tasks API 1の回数
  - 概要: 50ms以上かかっている処理。Chromeのみ

指標のログデータ

これらのデータをパフォーマンスログデータと呼ぶことにして、定期的に計測したものを貯めておくことにしました。 このパフォーマンスログデータの変化を見れば、パフォーマンスが改善しているのかの判断材料の1つとなりそうです。 （これらのパフォーマンスログデータ特定のリポジトリに追加していきました）

ログデータは単なるJSONファイルで、それだとモチベーションが湧きにくい/わかりにくいため、簡単なビューアアプリを書いてそれを見られる状態にしました。



> 縦軸がコメントのFPSで、横軸は時間軸

定期的にイベントが発生して、FPSに変化を与えていることがわかります。

RUMの信頼性

視聴中における体験を良くするのが目的なので、指標は明確に定めることが難しいです。 視聴中に継続的に取る値として次の項目を取得し、その傾向を参考程度にしかならない。 環境に左右されやすいRUM（Real User Measurement）なので、強く判断材料にはしにくいです（データ量がある程度必要）。

そのため、結局はボトルネックとなっているところを明確に見つけてそこに関するマイクロな改善を重ねることで、最終的な目的を達成する必要があります。

RUMと異なりSynthetic Monitoringといった環境が固定され定点できる手法ならば、ブレが起きにくいのでより指標に向いています。 一方、閲覧中（視聴中）の指標として扱うのが難しいです。

- 計測の種類 - Catchpoint | 株式会社Spelldata

RAILモデル

何かした時に反応が100ms以内、アニメーションが10ms以内、アイドル時の処理は50ms以上以内のブロックにする（long task）、Loadは1000ms以内などを指標を定めたRAILモデルなどがあります。

パフォーマンスの計測の仕方

> 計測できないなら安易に直さない

パフォーマンスの計測の仕方はさまざまです。 計測方法はボトルネックとなる部分（ネットワークやレンダリングといった箇所）によって異なります。

計測方法も大事ですが、計測する環境も重要です。

スロットリング

開発に使うようなマシンはハイスペックです。 パフォーマンスの計測を行う際は、必ずCPUスロットリングなどのスロットリングを行った状態で一度計測するべきです。

- CPUスロットリングを行う
  - `6x` 6倍遅い状態
- ネットワークスロットリングを行う

実際に利用した設定

この視聴中のパフォーマンス改善では次のような設定を利用して計測しています。 ネットワークスロットリングについてはランタイムではあまり関係ないものが多かったため特に設定していませんでした。

- 値の計測: Chrome
  - CPUスロットリング: 6x
  - ネットワークスロットリング: なし
- 動作の確認: IE/Edge/Firefox/Chrome/Safari

参考

- CPU Throttling - Chrome DevTools - Dev Tips
- さまざまなネットワーク状態でのパフォーマンスの最適化 | Tools for Web Developers | Google Developers
- モバイル開発に役立つJSデバッグ術 - Mercari Engineering Blog

また、Chromeのネットワークスロットリングはリクエストレベルのスロットリングになります。そのためパケットレベルのスロットリングをする場合は別のツールを使ったほうが安定します。

- lighthouse/throttling.md at master · GoogleChrome/lighthouse

実機確認

映像のデコードなどハードウェアの機能を使うものほど実機で確認するべきです。 VMのWindowsでは再現しないが、実機では再現するといった違いなどが発生します。

計測ツール

- Chrome DevTools
  - もっとも多機能で大部分をカバーできます
- `about:tracing`
  - グラフィックや入力レイテンシーなどレベルに近い情報を見られます

    - パフォーマンス計測に困らない！tracing活用術100 - Qiita
- ライブラリのデバッガ
    - 最近のライブラリにはたいていデバッガが着いています
    - facebook/react-devtools: An extension that allows inspection of React component hierarchy in the Chrome and Firefox Developer Tools.
- `performance.mark` 、 `performance.measures`
    - User Timing API: あなたの Web アプリをもっと理解するために - HTML5 Rocks
    - パフォーマンスまわりのAPIについて - Qiita
    - User Timing API - Web APIs | MDN

User Timing API

User Timing APIと呼ばれるものには `performance.mark` 、 `performance.measures` などがあります。 これらは、自分で指定した範囲をマーキングできます。 閲覧中（視聴中）のパフォーマンスというのは、機械的に取ることが難しいです。

そのため、 `performance.mark` 、 `performance.measures` を使ってアプリケーションに適した「指定範囲」を作ることがパフォーマンスの計測に役立ちます。

指標で作成したパフォーマンスログもこれらのAPIを使っています。

- performance.markでパフォーマンス計測する | Web Scratch
- performance.markにメタデータを紐付けできるライブラリを書いた | Web Scratch
- `performance.mark` with metadata is useful for Real user monitoring

また、最近はFluxやReduxなどイベントを発火してViewを更新するスタイルを取っていることが多いと思います。イベントが起点となっているときの利点として、何のイベントによって表示が更新されたかという関連付けが簡単なところにあります。

生放送の視聴ページではfluxとReactが使われていました。

この２つならイベント（FluxのDispatch）と表示の更新（Reactのrender）を関連付けて可視化することは簡単です。 VueやAlminなど最近のライブラリなら大抵何とかなります。

- Almin + React/Vue.jsのパフォーマンスプロファイルをタイムライン表示できるように | Web Scratch

イベントと更新の可視化

fluxのイベントとReactの更新の関係をUser Timing APIを使って可視化してみましょう。

といっても難しい話ではなく、Fluxがdispatchしたタイミングで `performance.mark` を貼り、Storeが更新されたタイミングもう一度 `performance.mark` を貼るだけです。 Reactは `?react_perf` （React 16からはデフォルト）を付けるだけで、自動的に `performance.mark` をつけてくれます。

- Profiling Components with the Chrome Performance Tab

FluxのDispatchの実装は次のような形にしました。

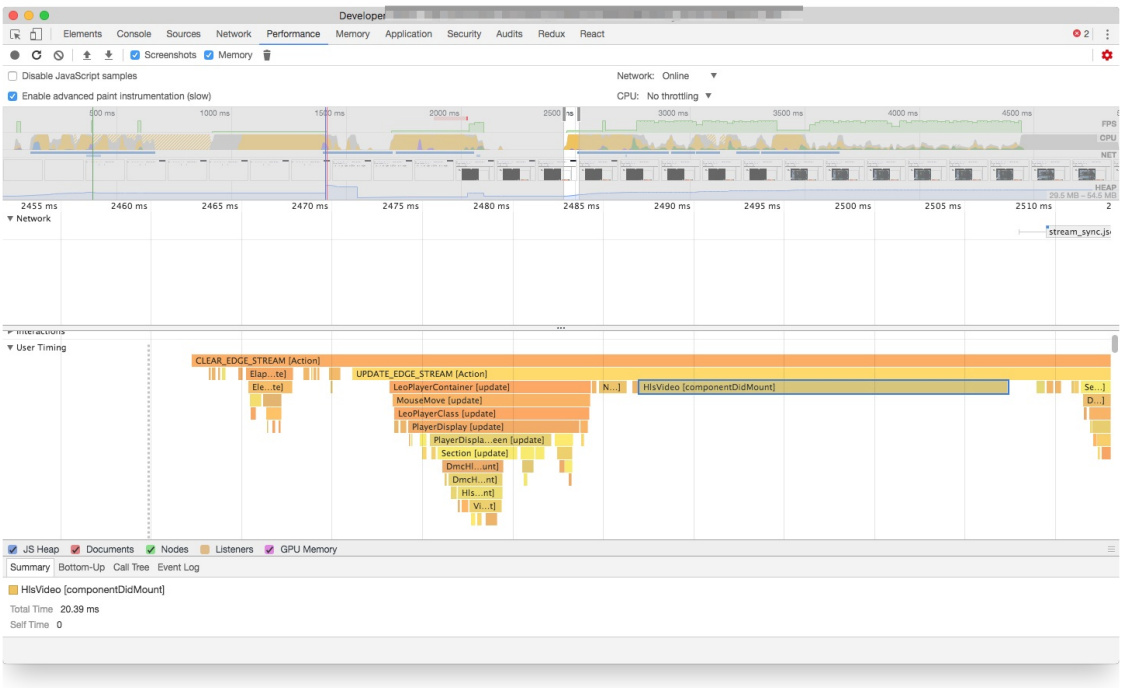
`Dispatcher` のシングルトンを作るときにUser Timingの処理を追加

```
import * as flux from "flux";
/**
 * アクションを発行した際に受け渡す情報.
 */
export interface ActionPayload {
    type: string;
}
```

```
/**
 * アクションを発行するもの.
 */
export const Dispatcher = new flux.Dispatcher<ActionPayload>();

/**
 * 開発向けのFluxのDispatchロガー
 *
 * - Fluxでdispatchしたアクションをコンソールへ出力
 * - タイムラインツールにアクションの実行を記録
 *  - 正確な実行時間は記録できないのであくまで参考
 *  - 実行時間はdispatchしてから次のrequestFrameAnimationが起きるまでの時間が記録されている
 *
 * URLに`?react_perfを付けることでActionとViewの処理の関係をタイムラインツールでみることができる
 *
 * - https://developers.google.com/web/tools/chrome-devtools/evaluate-performance/timeline-tool?hl=ja
 * - https://reactjs.org/docs/optimizing-performance.html#profiling-components-with-the-chrome-performance-tab
 *
 * 無効化する方法:
 *
 * 以下の環境変数と共に起動することで無効化される(production時は自動で無効化される)
 *
 * DISABLE_FLUX_LOG=1 npm start
 *
 */
if (process.env.NODE_ENV !== "production" && process.env.DISABLE_FLUX_LOG !== "1") {
    let requestAnimationFrameId: number | null = null;
    // 同期的に複数のPayloadが来ることがあるので貯める
    let currentPendingPayloads: ActionPayload[] = [];
    Dispatcher.register((payload: ActionPayload) => {
        // 1. performance.markの計測開始
        performance.mark(payload.type);
        if (requestAnimationFrameId) {
            cancelAnimationFrame(requestAnimationFrameId);
        }
        currentPendingPayloads.push(payload);
        // 現在はStoreのDispatcherTokenを取る方法がないため、非同期でちょっとだけ待つことで変化したStore一覧を取得している
        requestAnimationFrameId = requestAnimationFrame(() => {
            currentPendingPayloads.forEach(payload => {
                const type = payload.type;
                // 2. performance.markの計測終了
                const measureMark = `${type} [Action]`;
                performance.measure(
                    measureMark,
                    payload.type
                );
                // 計測に使ったmarkは削除する
                performance.clearMarks(type);
                performance.clearMeasures(measureMark);
            });
            currentPendingPayloads = [];
            requestAnimationFrameId = null;
        });
    });
}
```

これで、FluxがdispatchしたらそのActionの名前がChromeのタイムラインツールで見れます。 後は、Timeline ツールで単純に記録するだけで、「どのアクション」によって「どのコンポーネント」が更新されているかが可視化できます。



問題の糸口を探すには情報の可視化が大切です。

情報量の削減

Chromeの開発者ツールは情報量が多いため、逆に何が重要なのかがわからなくなる問題もあります。 そのような場合は、フィルタリングを工夫するか必要な情報だけを出すログツールを自分で書くのが簡単です。

大げさな仕組み化をしなければ既存の機能を乗っ取ったりしてログを得ることはできるはず。 この際に、エコシステムが大きなライブラリはそのようなツールもすでにあることが多いという利点があります。

Reactではさまざまなツールがありますが、必要に応じて使い分けることが大切です。

- why-did-you-update
- react-addons-perf（React 15限定）
- React Performance Devtool（React 16対応）
- amsul/react-performance: Helpers to debug and record component render performance
- nitin42/react-perf-devtool: A Chrome developer tool extension to inspect performance of React components.

CPUプロファイラを自作する例

- シンプルでかつ最高のJavaScriptプロファイラ sjsp を作りました！ — Webアプリケーションが複雑化する中でプロファイラに求められるものとは何か - プログラムモグモグ
- 45deg/node-sjsp: sjsp (Simple JavaScript Profiler) implemented in Node.js

Video要素やhls.jsのイベントを調べるツール

- azu/video-events-debugger: [WIP] HTML5 Video element events debugging tools.

無駄のタイプ

現状の機能やUIを維持しながら、パフォーマンスを良くすることは無駄を削ることとほぼ同義です。

生放送の視聴ページに関する無駄を削っていくことがパフォーマンスの向上につながります。 その"無駄"には幾つかの種類があると考えられます。

無駄の種類

ブラウザにはI/Oなどはないため主に次の要素が挙げられます。

- ネットワーク
- レンダリング
- スクリプティング

これはブラウザの開発者ツールのTimelineやNetworkパネルなどでみれます。

これらについては次のサイトや書籍が参考になります。

参考

- Chrome DevTools | Tools for Web Developers | Google Developers
- 開発ツール | MDN
- 超速！ Webページ速度改善ガイド
- Webフロントエンド ハイパフォーマンス チューニング
- Chrome Developer Toolsでパフォーマンス計測・改善 - Qiita

ネットワークにおける無駄

不必要なものを読み込まないのがもっとも効率的です。 余計なリソースを読み込まないことはファーストビューの表示に直接反映されるため、ページロードを短くするならばネットワークの無駄を探すのがもっとも効率的な改善です。 App Shell モデルなどはこの効率化をパターンしたものです。

- クリティカル レンダリング パス | Web | Google Developers

今回は目的で述べたように、視聴中におけるパフォーマンス改善なので省略します。

レンダリングとスクリプティングにおける無駄

生放送という性質上、視聴中に無駄となるものレンダリングやスクリプティングが多くの割合を占めます。 なぜなら、生放送の動画などは先読みキャッシュなどができないため、ネットワークは必要悪となりがちです。（映像の最適化などは別途必要）

視聴ページではReactが使われているので、reactでよくある問題を箇条書きします。

- 無駄な更新
  - これは同じ値にもかかわらず同じ結果に更新されているView
- 無駄な判定
  - これはshouldComponentUpdateの判定自体が不要なのにしなければならなくなっている場合
  - Parent > Child でParentの時点で受け取ったpropsが全く同じならChildではshouldComponentUpdateを判定自体しなくていい
  - ComponentのTree構造にも関係するので、トップダウンとボトムアップどちらも必要
  - トップダウン（上のコンポーネントで止める）と効果は高いが、相対的にトップでとまるものはあまり多く

ない
        - トップダウンで止めやすいようにDOM構造を変更するのも1つの手段
    - ボトムアップ（末端に近いコンポーネントで止める、結果的に止まればDOM APIを叩かない）で止めることは、効果は小さい範囲が広い
- 無駄なDOM API
    - `componentWillUpdate` や `componentDidUpdate` などでのDOM APIを叩く
    - 主にstyle操作やlayout操作に関係あるものをここでやるとコストが高い
    - たとえば `componentDidUpdate` でelement.getBoundingClientRectを叩くなど
        - CSS Triggersに載っているものを操作するDOM APIを叩くときは注意が必要
        - 方針としては、"更新したからLayout情報を取る"ではなく"更新されたからLayout情報を取る"にする
        - IntersectionObserver や `scroll` イベントなどイベントを使ってLayout情報を更新を行う
        - Viewをrenderしたからと言ってそのタイミングでLayoutが更新されるとは限らないので、イベント駆動にして最小限にする（基本Layoutコストは高い）
- 毎回作られる関数のハンドラ
    - propsで渡すハンドラを毎回作ってしまう問題
    - propsの関数同士の比較が毎回 `false` となってしまう（PureComponent）
    - shouldComponentUpdateの実装で関数同士は常にtrueという手もあり
    - React.js pure render performance anti-pattern – Esa-Matti Suuronen – Medium
    - 検知方法
    - TSLint: tslint-reactの `jsx-no-bind` and `jsx-no-lambda
    - ESlint: react/jsx-no-bind

修正例

実際の生放送の視聴ページに関する問題や修正についてをまとめて行きます。

多くの場合、次のような手順で「問題を発見して修正する」を繰り返しています。

1. 観測できる環境を作る
2. 観測する
3. 修正/計測方法を考える
4. 計測する
5. 修正する
6. 計測する

問題を修正するときにその修正の影響が全体にでることはすくないです。 そのため、基本的なマイクロな計測方法を考えて、修正時の前後の結果を提示します。

Note: 再現性

問題の再現性を見つけることはとても重要です。 再現性を見つけて初めてちゃんと観測できるようになります。

問題を見つけたときに、次のものを取っておくと修正に役立ちます。

- 再現手順
- スクリーンショット または 動画
- Timeline ツールのプロファイル

観測した時点で修正方法や問題の原因がわからなくても、Timeline ツールのプロファイルを取っておくと後から確認できるため役立ちます。 （プロファイルにはスクリーンショットも含まれるため描画の問題も再現できます）

生放送の問題を見つけるときは、できるだけプロファイルを取ることを意識して問題を発見していました。 修正後に前後のプロファイル見比べることでも、計測の参考になります。 記録したプロファイルをは適当なリポジトリを作成し、そこに状況と一緒に保存しておくようにしていました。

プロファイルには `performance.mark` や `performance.measure` の記録も残ります。 今時のフレームワークならこれらのAPIを使ったパフォーマンス計測もできるため、合わせて記録しておくことを推奨します。

- React: Profiling Components with the Chrome Performance Tab

`componentDidUpdate`でのReflowの修正

Reactではコンポーネントが描画/更新し終わると`componentDidUpdate`のライフサイクルメソッドが呼ばれます。 このとき、`componentDidUpdate`でReflowが発生するということは、描画が更新されるたびにレイアウト情報を計算し直さなければなりません。

> レイアウトの算出 各要素に割り当てられたスタイル情報をもとに、それぞれの要素がどの ような位置関係で配置されるのかを決定するのがレイアウトの算出処理で す。これが行われないと、要素がどんな位置にどんな大きさで配置されて いるのかはわかりません。 この処理の呼称はレンダリングエンジンによっ て異なります。Chrome の Blink や Safari の WebKitではこの処理を単にレイアウト(Layout)と呼び、Firefox の Gecko ではリフロー(Reflow)と呼びます が、本書ではレイアウトの算出と総称します。 JavaScript から Element#getBoundingClientRect() メソッドを実行して要素の矩形情報を取得したり、offsetTop や offsetWidth などのプロパティを 参照したりすることは頻繁にあります。 これらもレイアウトの算出処理によって得られた値を取得するための API です。このようなレイアウトに関わる情報の更新や参照がJavaScript から頻繁に行われることでレイアウト 算出の処理が誘発され、UI のスムーズさを損なう大きい負荷を生み出してしまうことがあります。 超速！ Webページ速度改善ガイド

ペイント（描画）とレイアウトは多くのブラウザでは、レンダリングプロセスでそれぞれ別々の処理となっています。 表示内容は変わっていても（ペイントはされても）、表示サイズといった枠組（レイアウト）は変わっていない場合などは、レイアウトはする必要がありません。

`live-video-component`

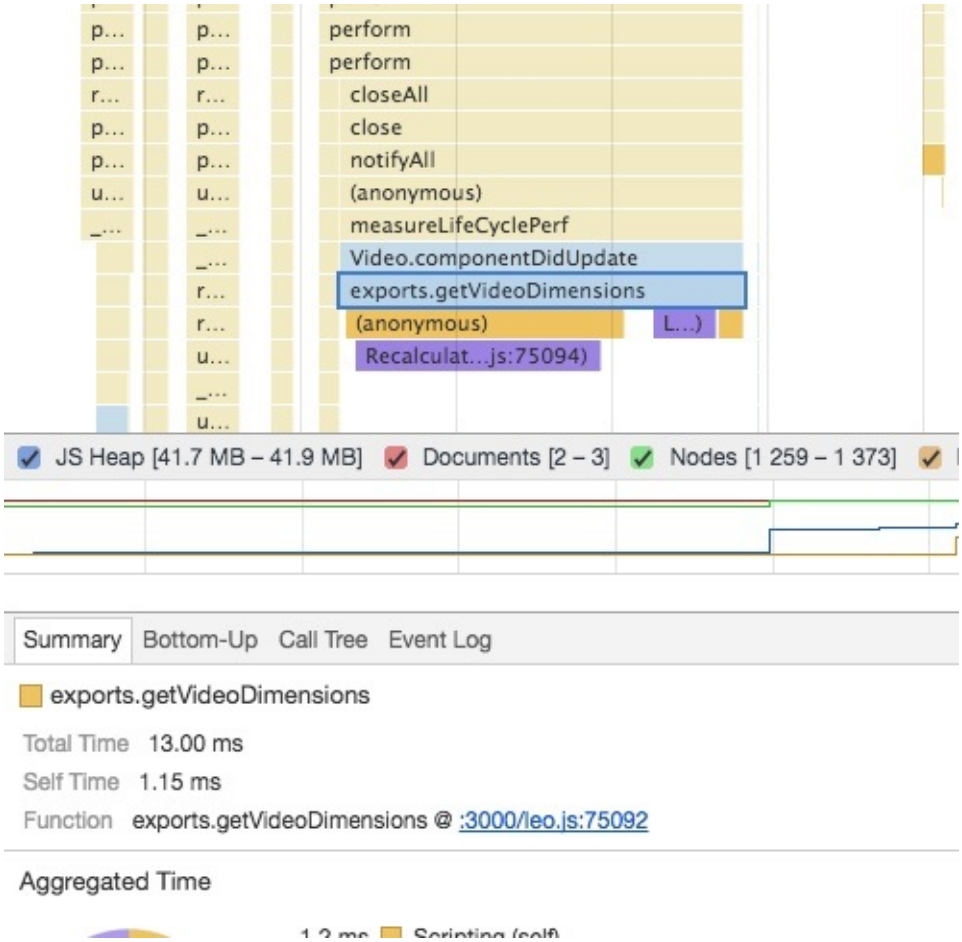
生放送では`live-video-component`という`Video`要素のラッパーコンポーネントを使っています。

観測

動画をずっと見ていると、動画が詰まったときのローディングアイコンがでるととても重たくなる現象を観測しました。

実際にそのタイミングでタイムラインツールのプロファイルを記録してみました。

このときにプロファイルを見てみると、`Video`コンポーネントの`componentDidUpdate`でレイアウトの再計算が発生していることに気づけます。



13msのうち殆どを占めている処理で1Frame分（16ms）程度かかっているので体感できる程度のカクツキがおきていました。

問題

このコンポーネントが `componentDidUpdate` で、Reflowを起こしているため、些細な更新であっても、重たいレイアウト処理が同時に発生しています。

このレイアウト処理の原因は `getVideoDimension` にあることはタイムラインからもわかります。

```
/**
 * 実際に表示される映像領域の寸法情報を取得
 */
export const getVideoDimensions = (video: HTMLVideoElement): VideoDimensions => {
    const aspectRatio = video.videoWidth / video.videoHeight;
    const elementWidth = video.clientWidth; // widthは%が入ってくることがあるのでpxが取れるclientWidthを使う
    const elementHeight = video.clientHeight;
    const elementRatio = elementWidth / elementHeight;
    const displayWidth = elementRatio > aspectRatio ? elementHeight * aspectRatio : elementWidth;
    const displayHeight = elementRatio < aspectRatio ? elementWidth / aspectRatio : elementHeight;
    return {
        videoWidth: video.videoWidth,
        videoHeight: video.videoHeight,
        aspectRatio: aspectRatio,
        displayWidth: displayWidth,
        displayHeight: displayHeight
    };
};
```

`getVideoDimensions`の中で、`clientHeight` / `clientWidth`など要素のサイズを再計算が必要なDOMプロパティに触れていることがわかります。

これらのプロパティはレイアウトを発生させるプロパティとして知られています。 次のページのそれらのプロパティのまとめられています。

- https://gist.github.com/paulirish/5d52fb081b3570c81e3a
- CSS Triggers

このプロパティを`componentDidUpdate`で参照しないようにできればこの問題は修正できます。

修正方針

この`getVideoDimensions`の用途はVideoコンポーネントのサイズが外的要因で変化した時に、サイズを取得して追従するために利用していたようでした。コンポーネントが更新されるたびに、サイズを取得しているというのが主な問題です。 コンポーネントの更新とコンポーネントのサイズが変わるかは別であるため、サイズ変化のイベントを監視して追従すべきです。

ブラウザのDOM APIには要素のサイズを変化を検知する方法があります。 そのため、描画更新の度(`componentDidUpdate`)にレイアウトの再計算を行うよりも、そのイベントに反応してレイアウトを計算する方が効率的です。

Note:

要素の大きさの変化を見るなら、ResizeObserverですが、まだ策定段階のAPIです。

- ResizeObserver

4.2 Element Resize Detectionには既存のAPIを使って要素のサイズ変化を検知する方法がまとめらています。

修正


perf(video): getVideoDimensionsを呼び出す回数を削減 #118

Merged live merged 3 commits into NicoliveComponent:master from live:resize-observe-onChangeVideoDimensions on 22 N

Conversation 4 Commits 3 Checks 0 Files changed 5

live commented on 20 Nov 2017 • edited Edited 1 time live edited on 22 Nov 2017 Member

概要

問題点

#116

プロファイルデータ
/getVideoDimensions.json

clientHeight/clientWidthはreflowの原因となるため、必要のないときに触るとコストが大きい
https://gist.github.com/paulirish/5d52fb081b3570c81e3a
そのため、 componentDidUpdate の度にreflowが発生していた。

大体: 5-16ms程度 on Chrome が発生していた

具体的には次のような修正を行いました。

- `componentDidUpdate` で `getVideoDimensions` をしていたのは、ブラウザフルにおいて要素のサイズの追従が必要かどうかのためだった
- つまり、要素のサイズが変わったタイミングで `onChangeVideoDimensions` を呼び出すことができれば `componentDidUpdate` から、 `getVideoDimensions` の呼び出しを削除することができる
- `react-notify-resize` を使い、要素のリサイズ時に `getVideoDimensions` を呼び出す実装に変更した

計測

修正後、Videoコンポーネントに渡すpropsを変化させた時のタイムラインプロファイルを取り、Videoコンポーネントの更新時にReflowが発生しなくなったことを確認できた。

フルスクリーン時に描画コストの問題の修正

観測

プレイヤーのフルスクリーン化、フルスクリーン解除時に音声が途切れることがありました。 どうやら、プレイヤーのフルスクリーン化、フルスクリーン解除時に大きな処理が行われているため、クライアントサイドの映像のデコードが遅れることがあるようだった。

フルスクリーンモードは一般に多くの要素が表示/非表示が切り替わるため負荷が高い処理ということが知られています。

問題点

- フルスクリーン化する際に非表示にする要素すべてunmountしている
- フルスクリーン解除する際にunmountした要素をすべてmountしなおしている

これにより、フルスクリーン化/フルスクリーン解除たびにPlayerDisplay（プレイヤーの動画やコントロール部分）以外をすべて再描画するのと同じことがおきていました。

また、要素を再描画する際に、componentDidUpdateでLayoutを行うプロパティを参照しているコンポーネントがいくつかありました。

これは次の問題と同じ原因です。

- `componentDidUpdate` でのReflowの修正

結果として、フルスクリーン時に再描画 -> 再レイアウト -> 再描画 -> 再レイアウトというように描画の負荷が蓄積していることがわかりました。

おさらい: componentDidUpdateでのレイアウト問題

componentDidUpdate()はrender後に呼ばれるライフサイクルメソッドです。 レンダリング後に重たい処理といえば、スクロールや要素のリサイズといったレイアウトを誘発する処理です。

基本的にcomponentDidUpdate()で重たい処理をやるのは向いていないです。 なぜならrenderする度に `componentDidUpdate` は呼ばれるため、不用意なレイアウトを誘発することがあるためです。

そのため、 `componentDidUpdate` でレイアウトを行うのではなく、必要なタイミング（イベント）で行うことで最小のコストでレイアウト処理を実装します。

リサイズイベントが起きたから、要素をリサイズするといったようにイベント駆動で処理することで負荷が軽減できます。もちろん大量のイベントが発生すると問題となるため、イベントを間引くや別のタイミングでまとめる工夫も必要になる場合があります。

少なくとも、 `shouldComponentUpdate` が完全に管理できていない状況においては `componentDidUpdate` でレイアウトするのは問題となりやすい箇所です。

- `componentDidUpdate` でのReflowの修正

componentDidUpdateのLayout問題の見つけ方

`componentDidUpdate` でLayoutが行われているかは開発者ツールのTimelineツールで `componentDidUpdate` の負荷を見ることで調べられます。

具体的な場所は次のような方法で調査しました。

ボトルネックの見つけ方

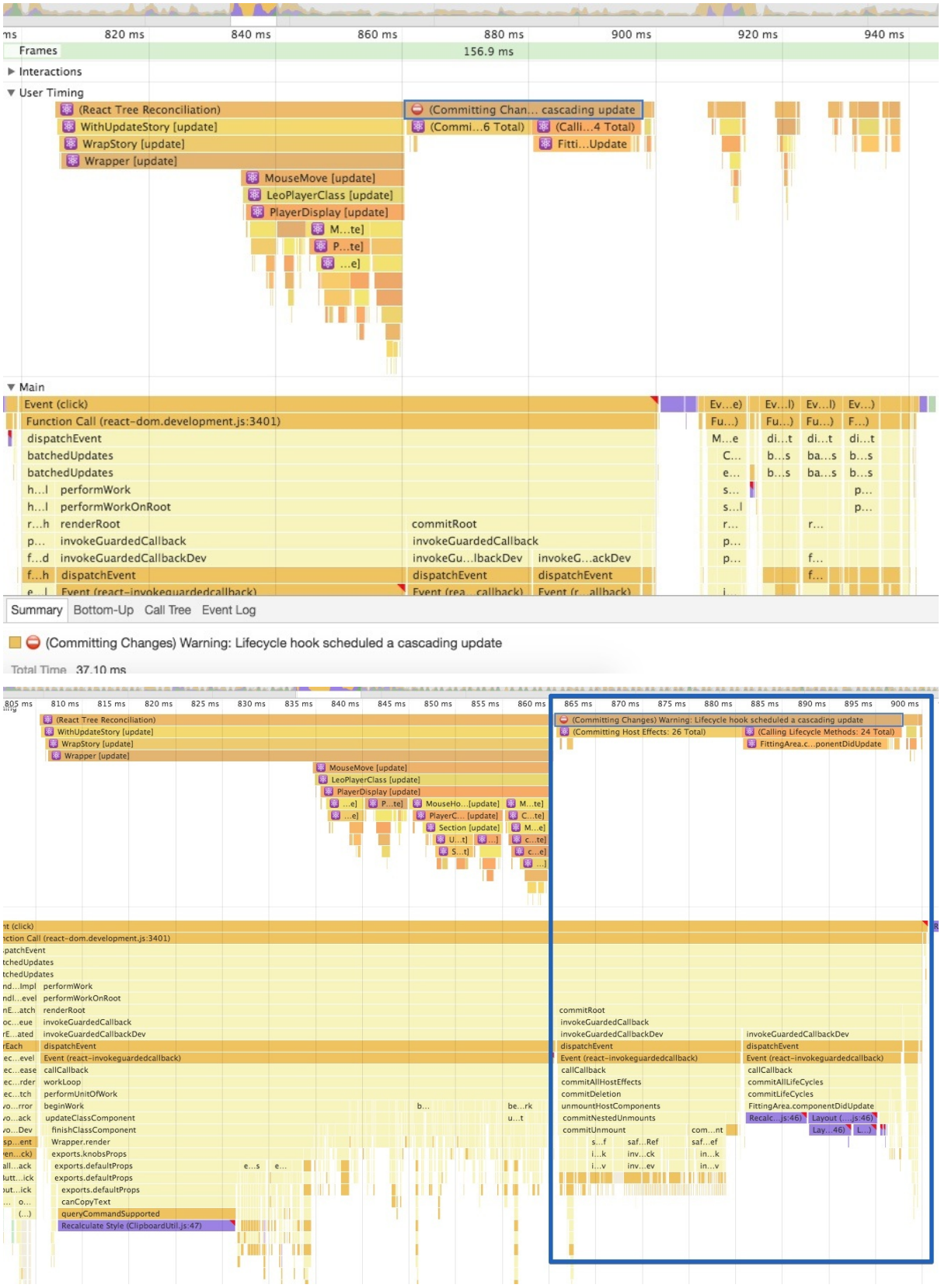
1. componentDidUpdateで重たい場所を見つける

React 16ならばTimelineに次のような表示が出ている箇所が怪しい箇所です。
（長い `componentDidUpdate` の処理の場合に表示される)

> (Committing Changes) Warning: Lifecycle hook scheduled a cascading update

- React 16 : componentDidUpdate Warning: Scheduled a cascading update · Issue #834 · reactjs/react-redux
- React 16 : componentDidUpdate Warning: Scheduled a cascading update · Issue #11987 · facebook/react
- performance - React 16 : componentDidUpdate Warning: Scheduled a cascading update - Stack Overflow

実際の例:



紫色のLayoutやRecalculate Styleに赤色の警告がでているなら重たい処理です。

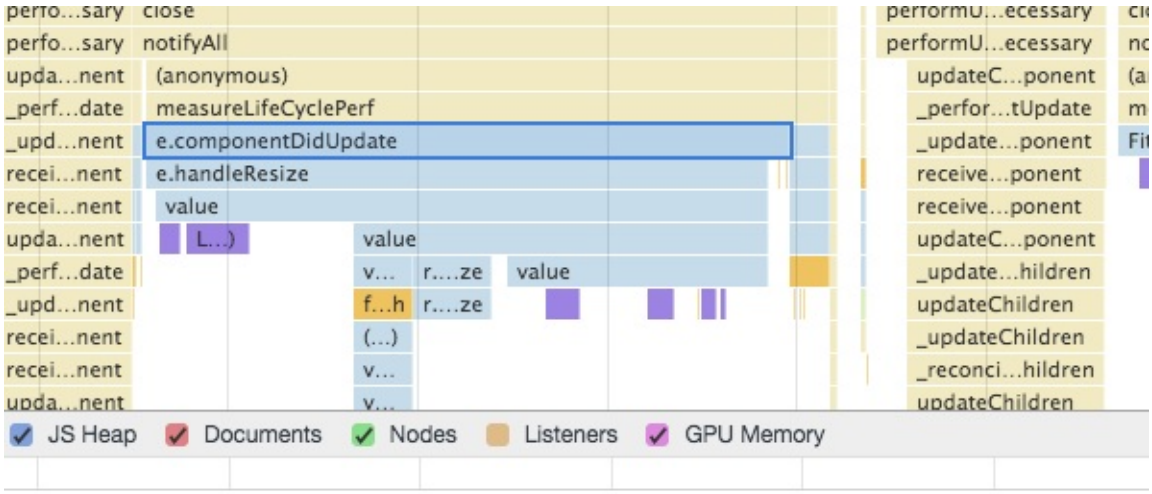
> Warning Forced reflow is a likely performance bottleneck. Total Time 6.58 ms

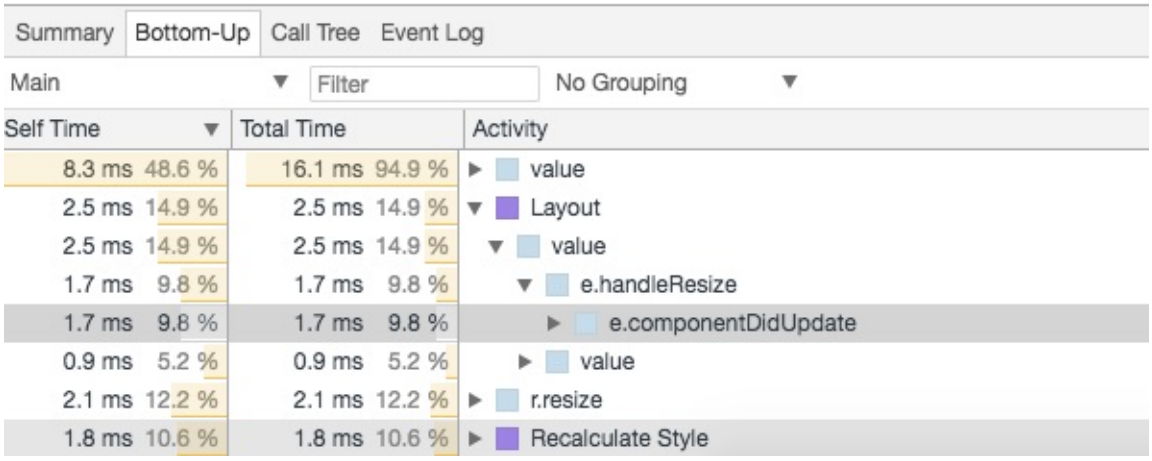
実際の処理時間以上に体感に影響を与えることがあります。

2. Reflowを起こしている箇所を見つける

componentDidUpdateで起きてることはわかっているので明白ですが、Timelineの処理時間から逆探知できます。

1. Reflow（紫色）ブロックの親ブロックを選択する
2. "Self Time"でソートする
3. 大体LayoutやRecalculate Styleが上位に来る
4. その中を掘っていくと、 `コンポーネント名.componentDidUpdate` が見つかる
5. このコンポーネントがReflowを起こしている直接的な要因です





修正

フルスクリーン時にReflowが発生することがわかったため、該当箇所をそれぞれ修正しました。

それぞれ `componentDidUpdate` ではなく、 `resize` イベントでレイアウトの再計算を行うようにしました。 こうすることで、描画のタイミングではなく、Resizeのタイミングで再計算を行うので無駄が少なくなりました。

perf(FittingArea): フルスクリーン時のパフォーマンス改善 #1130

Merged live merged 3 commits into `NicoliveComponent:react16` from `live:perf-fitting-area-layout` on 15 Jan

Conversation 1 | Commits 3 | Checks 0 | Files changed 3

live commented on 10 Jan • edited Edited 1 time live edited on 15 Jan — Member

概要

[redacted] と同じですが、
こちらは `FittingArea` に関する修正

問題点

フルスクリーン化、フルスクリーン解除する際に数回に分けて**Layout**の計算が行われている。
PlayerViewComponentにも問題があるが、NicoliveViewComponentの `FittingArea` も `componentDidUpdate` にて描画情報を取得する問題がある。
これによりrenderするたびに描画の再計算を行うため、フルスクリーン化する過程で再計算、完了して再計算と無駄な**Layout**(再計算)が発生している。

プロファイル

- LeoPlayerでフルスクリーン操作を行った時のデータ
- live/profile-live-2017-11

Reviewers: No reviews
Assignees: No one—as
Labels: None yet
Projects: None yet
Milestone: No mileston
Notification

perf(PlayerDisplayScreen): レイアウトの再計算をresizeハンドラで行うように変更 #115

Merged live merged 3 commits into `react16` from `fix-PlayerDisplayScreen-componentDidUpdate` on 16 Jan

Conversation 1 | Commits 3 | Checks 0 | Files changed 3

live commented on 10 Jan • edited Edited 1 time live edited on 16 Jan — Member

概要

問題点

[redacted] と同じですが、
こちらは `PlayerDisplayScreen` に関する修正です。

フルスクリーン化、フルスクリーン解除する際に数回に分けて**Layout**の計算が行われている。
PlayerViewComponentでの問題も同じように `componentDidUpdate` でLayoutを触ってDOMを更新している。
更新内容(`adjustScreenDimensions`)を見ると、基本的に領域が変更されたからそれに合わせるような内容だった。なのでリサイズ時に領域を変更すれば対応できる。

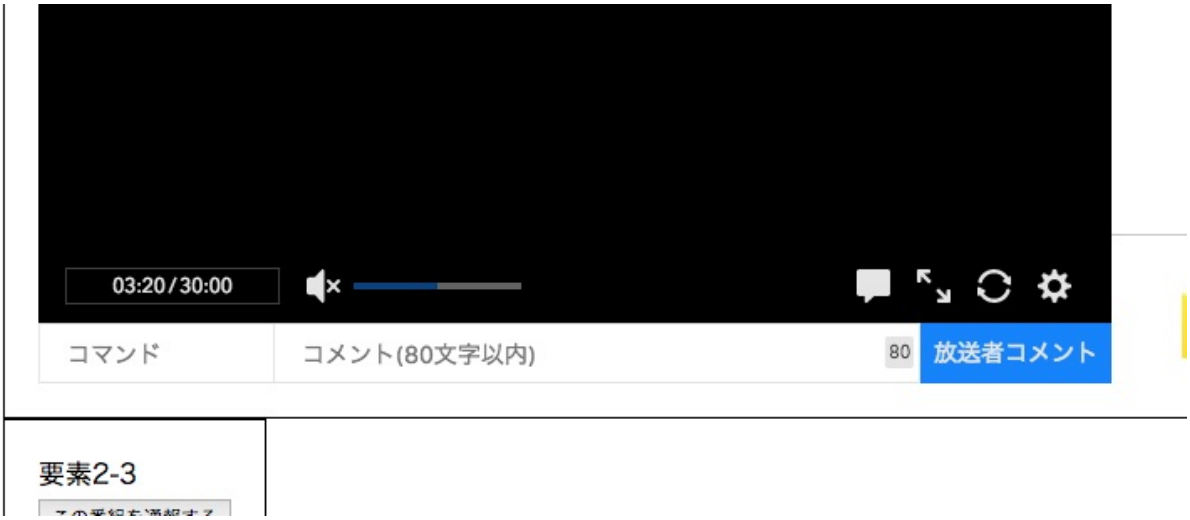
`NotifyResize` を使いリサイズ発生時に、 `adjustScreenDimensions` を適応するようにした

プロファイル

- LeoPlayerでフルスクリーン操作を行った時のデータ
- live/profile-live-2017-11

Reviewer: No revie
Assignee: No one a
Labels: None yet
Projects: None yet
Mileston: No miles
Notificat
You're re because

入力欄の更新コストの改善



コメントを投稿する入力欄は、文字入力するたびに更新します。 そのため、`<input onChange={更新処理} />` に安易な更新処理を書くととても重たくなる場合があります。

観測

視聴ページに主な入力欄は次の2つがあります。

- コメント入力欄
  - 1文字ずつユーザーが入力する
- コマンド入力欄
  - コマンドパレットから選ぶ
  - あまり大きな問題にはならない

実際に視聴ページでコメントを入力してみると、かなり広範囲が更新されていることがわかりました。
（光っているところが更新されている範囲）

参考

ニコニコ動画の視聴ページにおけるコメント入力の更新範囲。 こちらは、入力ごとに描画が更新される範囲が限定的になっていることがわかります。

問題

この更新範囲の広さの原因は、コメント欄に1文字入力する度、View全体を更新する処理が走っているためです。

1. FluxのDispatch
2. Storeを更新
3. StoreからView向けのPropsの作成
4. Viewの更新（全体を更新を試みる）

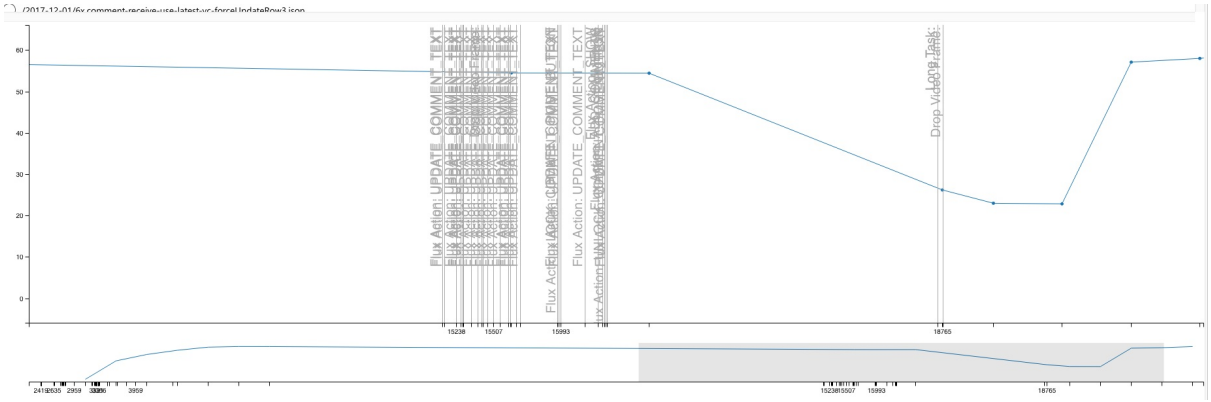
Reactの `input` 要素の更新方法には大きく分けて２つあります。

- 常にprops経由で更新するパターン
- コンポーネントのstateに値を持って更新するパターン

問題となっている実装は前者のprops経由でしか更新しないパターンになっていました。

- Forms - React
- Uncontrolled Components - React

計測時に取得したパフォーマンスログ:



上記のグラフの例を見ると、コメント入力のイベント（1文字ごとに発生）が発生後にFPSがガクッと下がっていることがわかります。

修正方針

この問題を修正する方法としては幾つかの選択肢があります。

1. Viewの中ですべて処理を終わらせる（ViewのStateで入力内容を管理）
   - Controlled Inputs
   - `<input type="text" value={this.state.value} onChange={this.handleChange}/>`
   - 投稿ボタンを押したときにViewから入力内容を取って投稿する
   - これはviewのなかにロジックを書く
2. Viewの更新範囲を限定的にできるような仕組みを作る
3. ContainerからViewのStateを直接操作するOperatorのようなものを作る
   - 具体的には入力欄を更新する `updateText` とか `updateScore` とかそういうのが必要
   - Viewの値(`state`)を半分直接操作できるようにする

ここまでの内容をチームと話し合い、方針 `3` で修正することにしました。

- 実際に話し合いで利用したスライド: コメント入力欄のパフォーマンス問題

修正

コンポーネント側がマウントした時に `operator` オブジェクトを取得できるような形にして、 `operator.updateText()` や `operator.getValue()` という形で直接更新できる機能を追加しました。 そしてそのコンポーネントを使って入力中は、直接更新するようにし、入力が終わったタイミングで一度全体を更新するような形にしました。

先程の修正方針の3と既存のFluxループの仕組みを合わせた形になっています。 これにより、入力中は入力欄だけが更新され、入力が終わったタイミング（debounceして終わったと判断する）でもう一度同期できます。

2種類の更新パターンを使ったハイブリット方式になります。

1. 入力中: コメント入力欄だけを更新
2. 入力後: 全体を同期させるように更新

こうすることで入力中の負荷の問題とコメントの文字数に応じた動作が解決できます。

計測

修正した結果、入力中に更新されるのがコメント入力欄だけに限定されるようになった。

修正前

修正後

参考

- Ah, interesting. – Con Antonakos – Medium

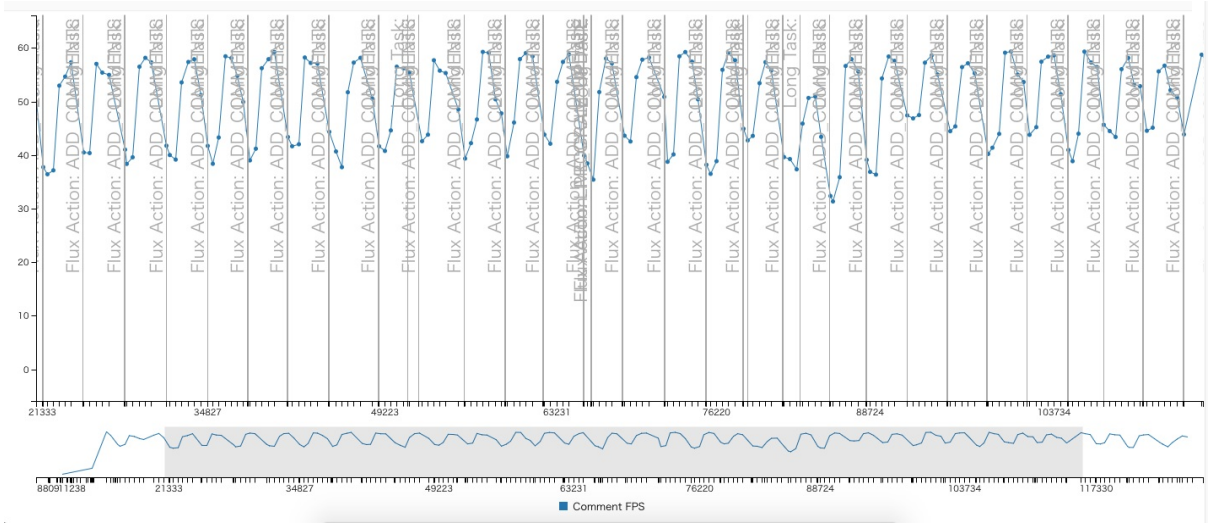
リストコンポーネントの修正

リストコンポーネントは多くの要素を扱う性質上レンダリングのボトルネックとなりやすいです。

生放送の視聴ページにもコメントパネルと呼ばれるリストを扱うコンポーネントがあります。

低スペックのデバイスでは、このリストコンポーネントにコメントを追加する処理が定常的に時間がかかっていることが観測できていました。

指標で作成したパフォーマンスログをみた結果、コメントを追加(`ADD_COMMENT` のFluxログ)が起きる度に、コメントのCanvasのFPSが下がっていることがわかります。



FPSの急激な変化は体感が悪いため、コメント（アイテム）をコメントパネル（リスト）に追加する時の処理は改善するべき項目だと判断できます。（緩やかな変化が望ましい）

リストコンポーネントは最適化のために表示範囲の要素だけを描画する仕組みなどがすでに実装されていました。

- バーチャルレンダリング
- Webフロントエンド ハイパフォーマンス チューニング

そのような処理の軽減があっても、コメントを追加する度に数十個の要素が追加/削除されてたりしているため、スペックの低いデバイスではボトルネックとなります。

何が原因で処理が重くなっているのかを調べつつ、それらを修正していきます。

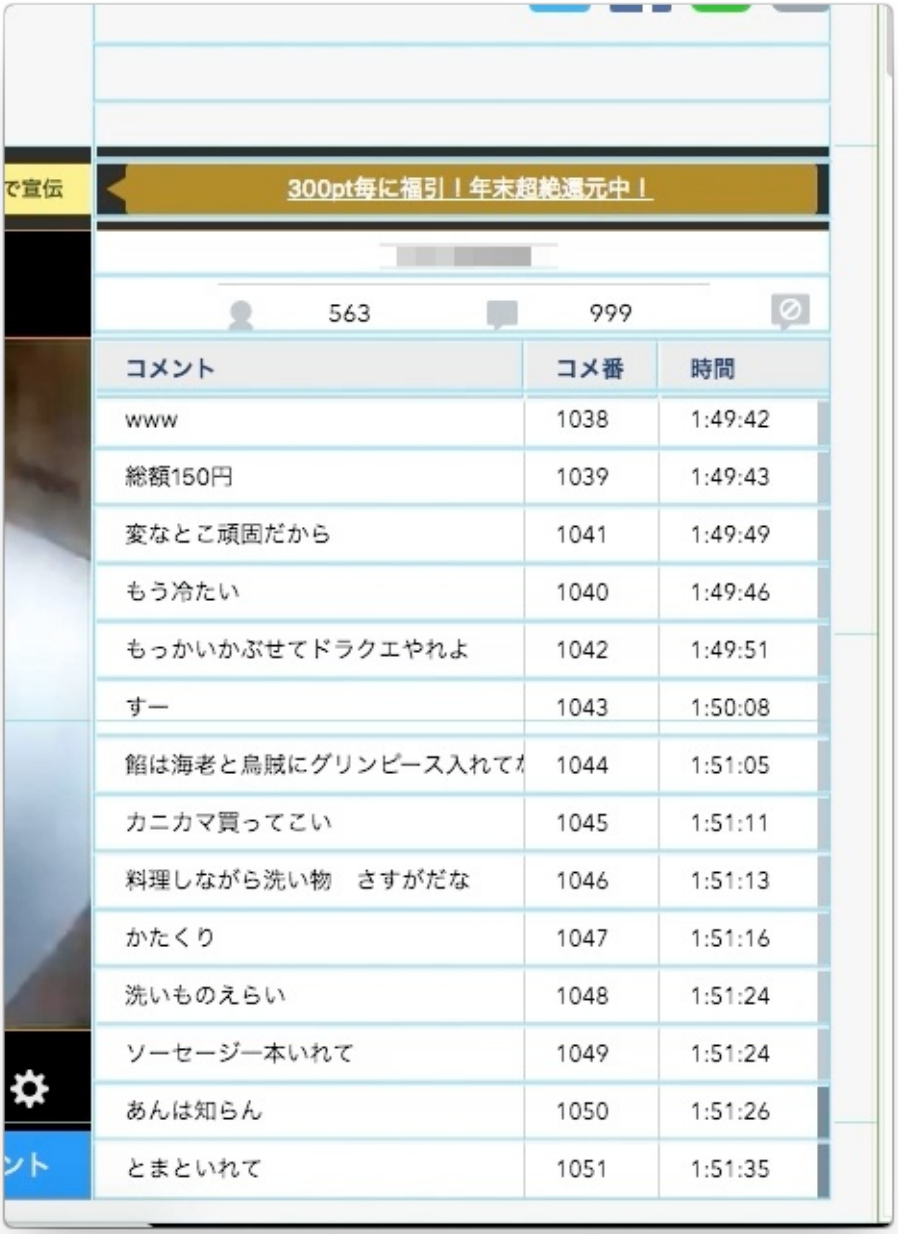
リストコンポーネントの `shouldComponentUpdate` の改善

コメントパネルは `DataGrid` と呼ばれる汎用的なリスト表示のコンポーネントで作られていました。

`DataGrid` は次のように1行ごとにアイテムを並べ、表示領域の上下の見えない空間は空のdiv要素を配置して高さを作るといった作りになっています。

DataGrid

次の動画のように `DataGrid` にコメントを追加すると、表示範囲外の描画も発生していることが分かると思います。


で宣伝
300pt毎に福引！年末超絶還元中！
563
999
コメント
コメ番
時間
www 1038 1:49:42
総額150円 1039 1:49:43
変なとこ頑固だから 1041 1:49:49
もう冷たい 1040 1:49:46
もっかいかぶせてドラクエやれよ 1042 1:49:51
すー 1043 1:50:08
餡は海老と烏賊にグリンピース入れてた 1044 1:51:05
カニカマ買ってこい 1045 1:51:11
料理しながら洗い物　さすがだな 1046 1:51:13
かたくり 1047 1:51:16
洗いものえらい 1048 1:51:24
ソーセージ一本いれて 1049 1:51:24
あんは知らん 1050 1:51:26
とまといれて 1051 1:51:35

これは高さを確保するための仕組みなので意図とおりです。 しかし、リストはリストアイテムの更新する時の処理を小さくできればそのまま `リストアイテムの数 * 改善した時間` 分だけ処理時間が減ります。 そのため、リストアイテムそれぞれの最適化をするモチベーションは大きいです。

観測

リストコンポーネントに無駄な処理がないかを詳しく調べるために、 `react-addons-perf` を使って無駄な更新が行われていないかを調べてみることにしました。

> 注記: `react-addons-perf` はReact 15でしか動きません。 React 16以降はreact-performanceや公式のプロファイラを利用します。

- Performance Tools - React
- Reactを使ったプロダクトのパフォーマンスを改善した話 | GMOメディア エンジニアブログ

`react-addons-perf` にはprintWasted()という、無駄な更新をテーブル表示してくれる機能があります。ここでいう無駄な更新というのは、Propsなどが同じrender結果が同じであったためVirtual DOMに吸収され結局何も変わらなかった更新のことを言います。

再現性がある操作で再現性がある結果を得るために、まずはどういう操作をした時の処理を記録するかを考えました。 `DataGrid` はコンポーネント単位で開発されていてStorybookというプレビュー環境で動かすことができるようになっていました。 そのため、このStorybookで特定の操作をした時の動作を観測していきました。

> StorybookでのDataGrid

操作内容

DataGridの典型的なユースケースはコメントを追加することであるため、次の操作をした時の処理を`react-addons-perf`で記録しました。

1. DataGridのStoryで10コずつコメントを末尾に追加する
2. これを15回繰り返す
3. ReactPerfAddonで計測結果を出す
    - `printWasted()`のWasted Timeが必要がない更新にかかっている時間

（改善前）計測結果

1回目

```
> Perf.printInclusive()
  Perf.printWasted()
```

ReactPerf.js:30

(index)	Owner > Component	Inclusive render time (ms)	Instance count	Render count
0	"WithUpdateStory"	590.94	1	17
1	"WithUpdateStory > WrapStor…	501.68	1	17
2	"WithUpdateStory > Chapter"	501.52	1	17
3	"WithUpdateStory > ExampleG…	494.17	1	17
4	"WithUpdateStory > Example"	487.6	2	34
5	"WithUpdateStory > DataGrid…	468.26	1	17
6	"DataGridWrapper > DataGrid"	424.25	1	19
7	"DataGrid > RowComponent"	216.17	150	343
8	"DataGrid > TableCell"	125.36	300	686
9	"WithUpdateStory > ExampleT…	10.34	1	17
10	"DataGrid > HeaderCell"	9.97	2	38

▶ Array(11)

ReactPerf.js:30

(index)	Owner > Component	Inclusive wasted time (ms)	Instance count	Render count
0	"DataGrid > RowComponent"	119.78	140	193
1	"DataGrid > TableCell"	70.43	280	386
2	"WithUpdateStory > Example"	18.02	2	17
3	"WithUpdateStory"	10.08	1	1
4	"WithUpdateStory > ExampleT…	8.84	1	16
5	"DataGrid > HeaderCell"	8.56	2	36
6	"WithUpdateStory > WrapStor…	7.21	1	1
7	"WithUpdateStory > Chapter"	7.18	1	1
8	"WithUpdateStory > ExampleG…	5.72	1	1
9	"WithUpdateStory > DataGrid…	3.9	1	1
10	"DataGridWrapper > DataGrid"	2.21	1	1

▶ Array(11)

2回目

```
> Perf.printInclusive()
  Perf.printWasted()
```

ReactPerf.js:30

(index)	Owner > Component	Inclusive render time (ms)	Instance count	Render count
0	"WithUpdateStory"	514.12	1	17
1	"WithUpdateStory > WrapStory"	424.8	1	17
2	"WithUpdateStory > Chapter"	423.07	1	17
3	"WithUpdateStory > ExampleGr…	418.44	1	17
4	"WithUpdateStory > Example"	410.99	2	34
5	"WithUpdateStory > DataGridW…	390.11	1	17
6	"DataGridWrapper > DataGrid"	360.74	1	19
7	"DataGrid > RowComponent"	183.82	150	343
8	"DataGrid > TableCell"	108.46	300	686
9	"WithUpdateStory > ExampleTa…	14.26	1	17
10	"DataGrid > HeaderCell"	10.39	2	38

▶ Array(11)

ReactPerf.js:30

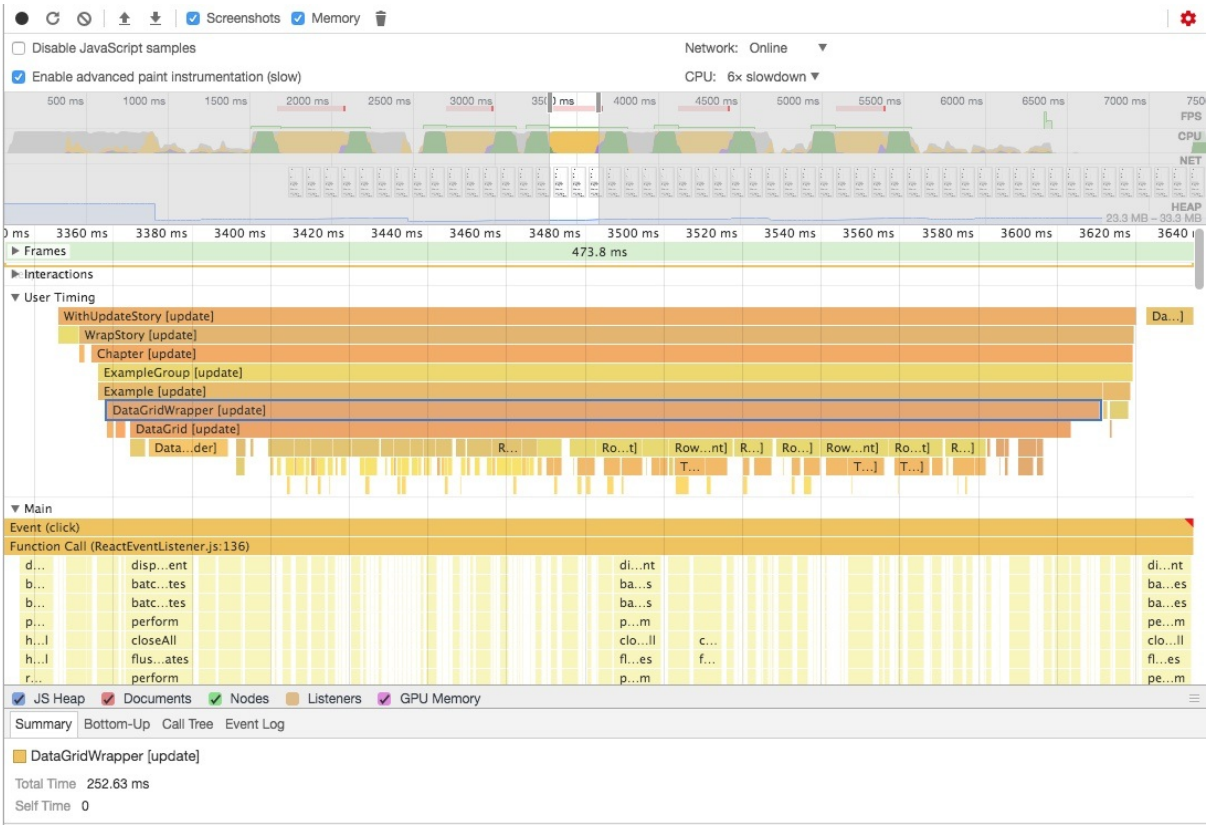
(index)	Owner > Component	Inclusive wasted time (ms)	Instance count	Render count
0	"DataGrid > RowComponent"	97.43	140	193
1	"DataGrid > TableCell"	56.25	280	386
2	"WithUpdateStory > Example"	22.51	2	17
3	"WithUpdateStory > ExampleTa…	12.52	1	16
4	"DataGrid > HeaderCell"	8.35	2	36
5	"WithUpdateStory"	8.05	1	1
6	"WithUpdateStory > WrapStory"	6.32	1	1
7	"WithUpdateStory > Chapter"	6.31	1	1
8	"WithUpdateStory > ExampleGr…	6.28	1	1
9	"WithUpdateStory > DataGridW…	4.71	1	1
10	"DataGridWrapper > DataGrid"	3.9	1	1

▶ Array(11)

```
< undefined
```

プロファイル結果

合わせてタイムラインツールでプロファイル結果を記録しました。

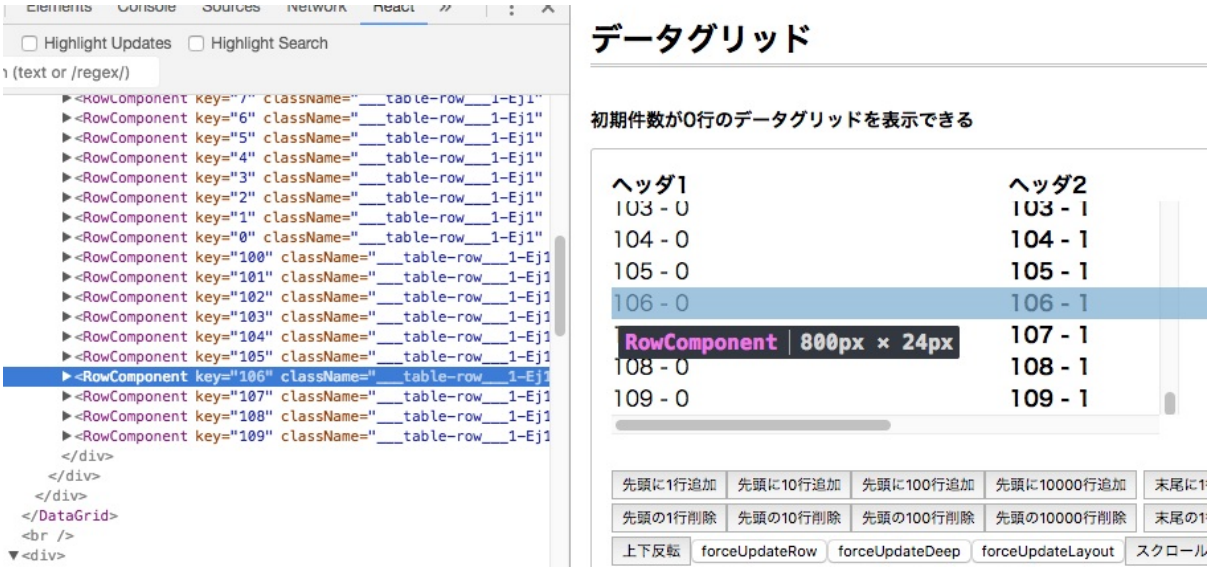


一度（10個追加）のDataGridのupdateにかかっている時間

- 252.63 ~ 300ms

結果から分かること

`RowComponent` という `DataGrid` の内のコンポーネントが無駄に更新されていることがわかります。（90-110ms程度が無駄になっている） `RowComponent` は、リストのリストアイテムのコンポーネントでした。



何が無駄なのかを具体的に調べてみると、次のことがわかりました。

- DataGridにコメントを追加する時、既存の `RowComponent` は再利用できるはず
- 実際には `RowComponent` は再利用されずに、追加のたびに同じpropsで再描画されていた

この `RowComponent` コンポーネントの無駄な描画を減らせば `無駄な行数 * コスト` が削減できることになります。

修正方針

無駄な再描画が起きないように同じpropsなら同じコンポーネントの結果を返せるように `shouldComponentUpdate` を正しく実装してあげればよさそうです。 リストのように巨大な配列がpropsに入る可能性がある場合は必ずshallow equalのような浅い比較で済むように気をつける必要があります。 そうしないと `shouldComponentUpdate` 自体の処理が重くなってしまって意味がなくなります。

参考:

- shallow-equal for Object/React props | Web Scratch

修正

`DataGrid` では、関数をpropsとして受け取り、その関数が `RowComponent` を作るという仕組みになっていました。 簡略化すると次のようなコンポーネントを返す関数を受け取り、内部でその関数を実行して `RowComponent` が作られています。

```
// `RowComponent`を作る関数
const generateRowComponent = (rowInfomation) => {
    return <RowCompoment {...rowInformation} />
}
// DataGridにその関数をわたす
<DataGrid generateRowComponent={generateRowComponent} />
// .. DataGridの中では ...
class DataGrid extends React.Component{
    render(){
        const アイテムの情報一覧 = this.props.アイテムの情報一覧;
        const items = アイテムの情報一覧.map(アイテム情報 => {
            return this.props.generateRowComponent(アイテム情報);
        });
        return <ul>
            {items}
        </ul>
    }
}
```

`RowComponent` を使いまわすようにキャッシュしたいのが目的であるため、 `generateRowComponent` が同じ引数なら同じ結果を返すようなメモ化をすれば解決できます。

```
const memorizedGenerateRowComponent = memorize(generateRowComponent)
// DataGridにメモ化した関数をわたす
<DataGrid generateRowComponent={generateRowComponent} />
```

しかし、できるだけReactの動作フローで解決したかったので、 `generateRowComponent` が返すコンポーネントをラップした `React.Component` を作ることにしました。 次のように `generateRowComponent` の関数をラップし、同じPropsなら同じ結果がキャッシュされるようにしました。

```
// `RowComponent`を作る関数
const generateRowComponent = (rowInfomation) => {
    return <RowCompoment {...rowInformation} />
}
// DataGridにその関数をわたす
<DataGrid generateRowComponent={generateRowComponent} />
// ここまでは同じ
// .. DataGridの中では ...
```

```
// generateRowComponentをラップしキャッシュできるコンポーネントを作る
class WrapperRowComponent extends React.Component{
    shouldComponentUpdate(nextProps){
        // ここでpropsの判定を正しくやればWrapperRowComponentは使いまわされる
    }
    render(){
        // 渡されたgenerateRowComponent()からコンポーネントを作って返す
        reuturn this.props.generateRowComponent(this.props.アイテム情報);
    }
}
class DataGrid extends React.Component{
    render(){
        const アイテムの情報一覧 = this.props.アイテムの情報一覧;
        const items = アイテムの情報一覧.map(アイテム情報 => {
            return <WrapperRowComponent アイテム情報={...アイテム情報} generateRowComponent={this.props.generateRow
Component} />
        });
        return <ul>
            {items}
        </ul>
    }
}
```

perf(DataGrid): DataGridのパフォーマンス改善 #1094

Merged live merged 20 commits into `NicoliveComponent:v6` from `live:perf-datagrid` on 1 Dec 2017

Conversation 33 Commits 20 Checks 0 Files changed 8

live commented on 29 Nov 2017 • edited Edited 1 time live edited on 1 Dec 2017 Member

概要

パフォーマンス調査と改善タスク
/issues/674 の一環

DataGridのパフォーマンスを改善するPRです。
何か壊してるかもしれないしれません(比較を厳しくした)

方針としては、RowComponentの更新コストが大きかったため、
RowComponentやその中身のCell `shouldComponentUpdate` を実装して無駄な更新を省くようにしました。

確認環境

- Chrome
- 6x CPUスロットリング

操作

- DataGridのStoryで10コずつコメントを**末尾**に追加する
- これを15回繰り返す
- その後 ReactPerfAddonで測定する。
  - スクショに2番目にある `printWasted()` の `Wasted Time` が必要がない更新にかかっている時間

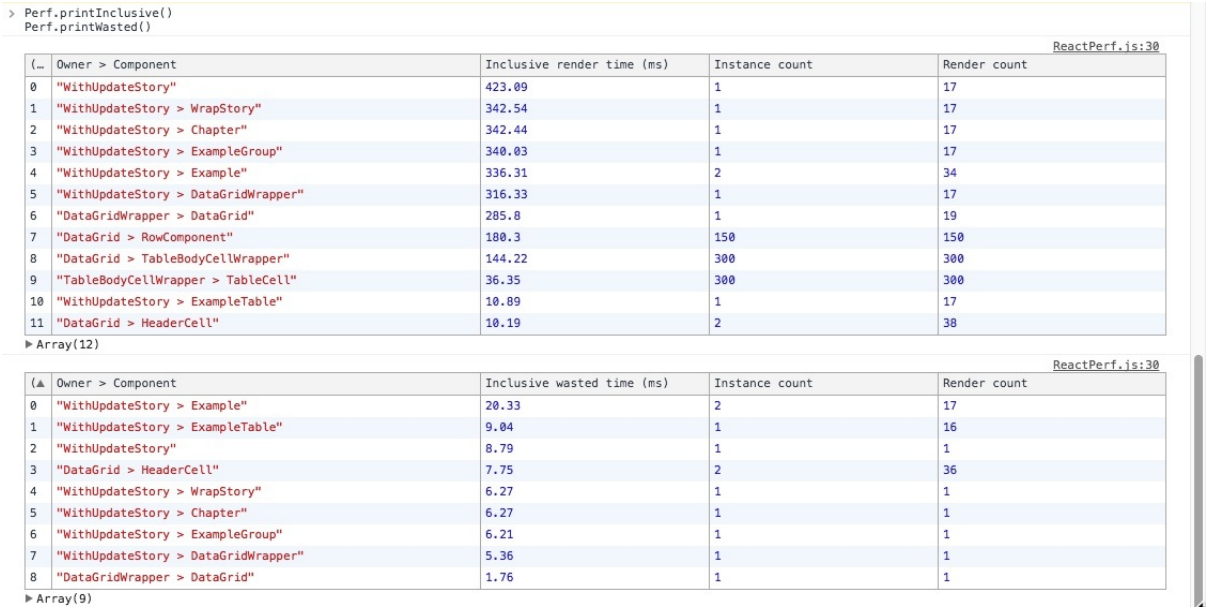
react addon perfについては次を参照(React 16で使えなくなる)

- Reactを使ったプロダクトのパフォーマンスを改善した話 | GMOメディア エンジニアブログ
- Performance Tools - React

計測

修正前と同じ手順で `react-addons-perf` を記録しました。

1回目



```
> Perf.printInclusive()
  Perf.printWasted()
```

ReactPerf.js:30

(…	Owner > Component	Inclusive render time (ms)	Instance count	Render count
0	"WithUpdateStory"	423.09	1	17
1	"WithUpdateStory > WrapStory"	342.54	1	17
2	"WithUpdateStory > Chapter"	342.44	1	17
3	"WithUpdateStory > ExampleGroup"	340.03	1	17
4	"WithUpdateStory > Example"	336.31	2	34
5	"WithUpdateStory > DataGridWrapper"	316.33	1	17
6	"DataGridWrapper > DataGrid"	285.8	1	19
7	"DataGrid > RowComponent"	180.3	150	150
8	"DataGrid > TableBodyCellWrapper"	144.22	300	300
9	"TableBodyCellWrapper > TableCell"	36.35	300	300
10	"WithUpdateStory > ExampleTable"	10.89	1	17
11	"DataGrid > HeaderCell"	10.19	2	38

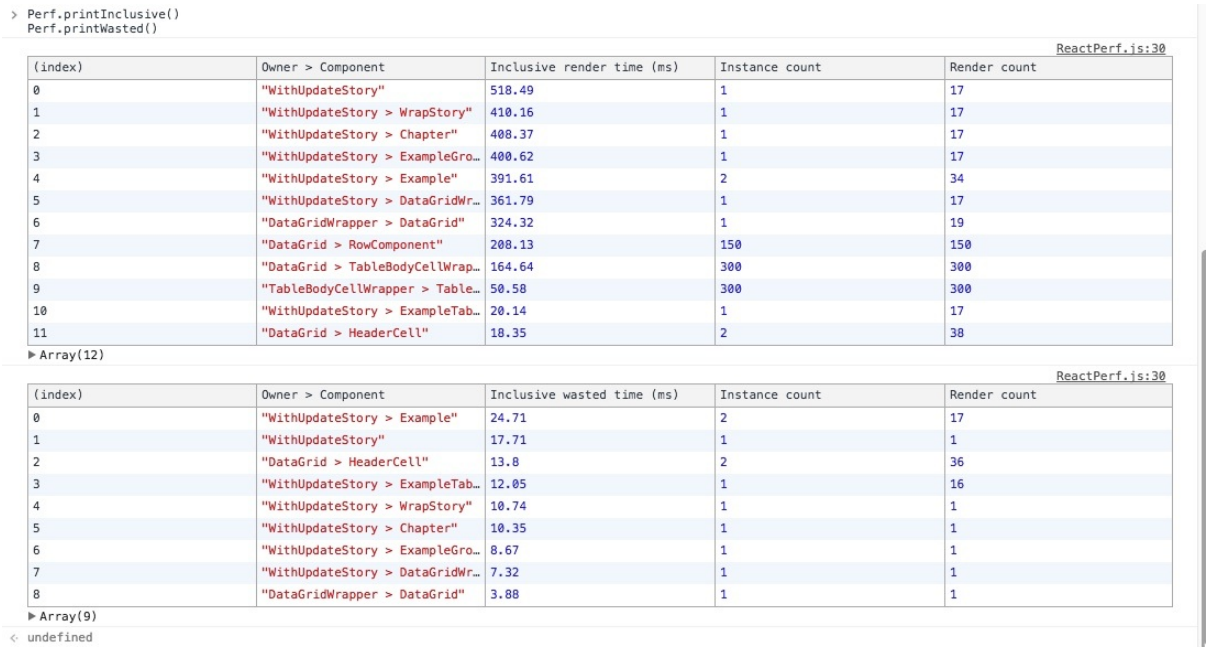
▶ Array(12)

ReactPerf.js:30

(▲	Owner > Component	Inclusive wasted time (ms)	Instance count	Render count
0	"WithUpdateStory > Example"	20.33	2	17
1	"WithUpdateStory > ExampleTable"	9.04	1	16
2	"WithUpdateStory"	8.79	1	1
3	"DataGrid > HeaderCell"	7.75	2	36
4	"WithUpdateStory > WrapStory"	6.27	1	1
5	"WithUpdateStory > Chapter"	6.27	1	1
6	"WithUpdateStory > ExampleGroup"	6.21	1	1
7	"WithUpdateStory > DataGridWrapper"	5.36	1	1
8	"DataGridWrapper > DataGrid"	1.76	1	1

▶ Array(9)

2回目



```
> Perf.printInclusive()
  Perf.printWasted()
```

ReactPerf.js:30

(index)	Owner > Component	Inclusive render time (ms)	Instance count	Render count
0	"WithUpdateStory"	518.49	1	17
1	"WithUpdateStory > WrapStory"	410.16	1	17
2	"WithUpdateStory > Chapter"	408.37	1	17
3	"WithUpdateStory > ExampleGro…	400.62	1	17
4	"WithUpdateStory > Example"	391.61	2	34
5	"WithUpdateStory > DataGridWr…	361.79	1	17
6	"DataGridWrapper > DataGrid"	324.32	1	19
7	"DataGrid > RowComponent"	208.13	150	150
8	"DataGrid > TableBodyCellWrap…	164.64	300	300
9	"TableBodyCellWrapper > Table…	50.58	300	300
10	"WithUpdateStory > ExampleTab…	20.14	1	17
11	"DataGrid > HeaderCell"	18.35	2	38

▶ Array(12)

ReactPerf.js:30

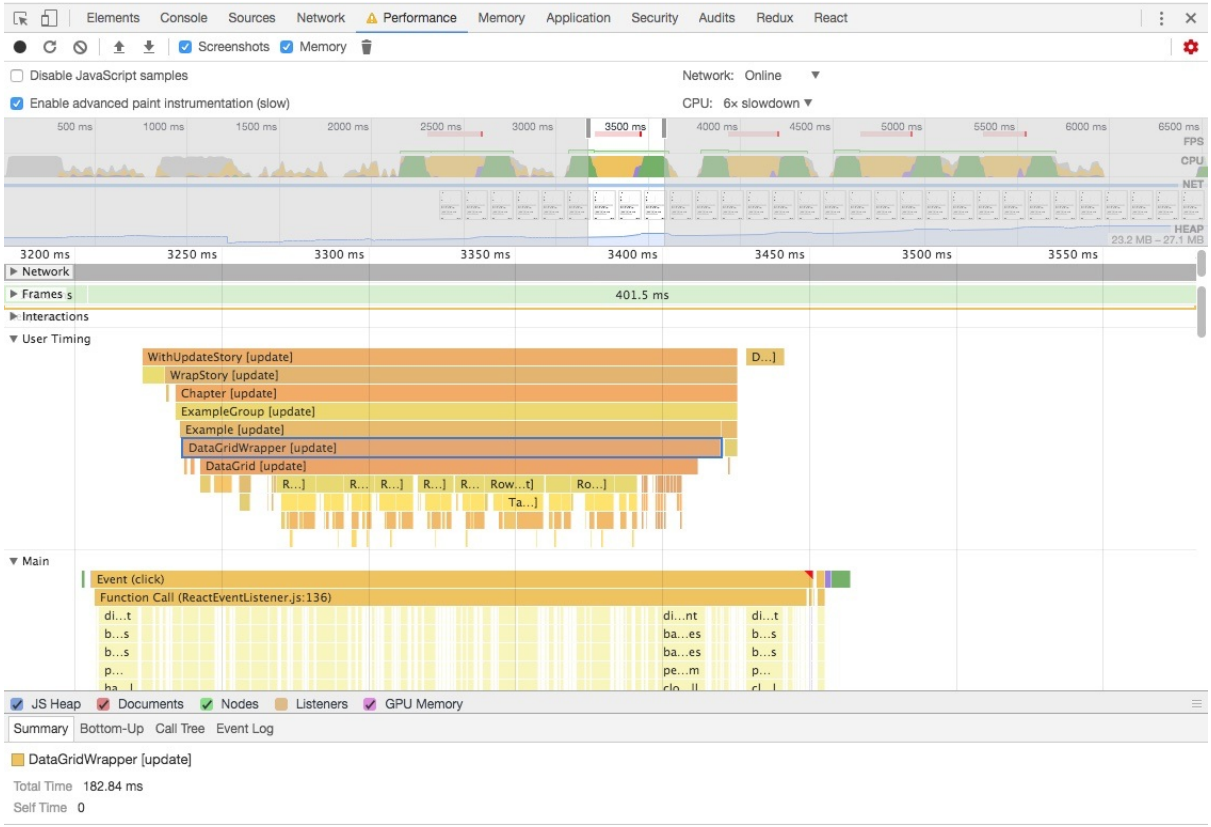
(index)	Owner > Component	Inclusive wasted time (ms)	Instance count	Render count
0	"WithUpdateStory > Example"	24.71	2	17
1	"WithUpdateStory"	17.71	1	1
2	"DataGrid > HeaderCell"	13.8	2	36
3	"WithUpdateStory > ExampleTab…	12.05	1	16
4	"WithUpdateStory > WrapStory"	10.74	1	1
5	"WithUpdateStory > Chapter"	10.35	1	1
6	"WithUpdateStory > ExampleGro…	8.67	1	1
7	"WithUpdateStory > DataGridWr…	7.32	1	1
8	"DataGridWrapper > DataGrid"	3.88	1	1

▶ Array(9)

← undefined

Wasted Timeから`RowComponent`が消えて、無駄な更新がなくなっていることがわかります。

プロファイル



一度（10個追加）のDataGridのupdateにかかっている時間

- 150˜220ms

修正前に比べると更新処理が100ms程度改善していることがわかります。

- Before: 250˜300ms
- After: 150˜220ms

リストコンポーネントへの追加処理の修正

リストコンポーネントの `shouldComponentUpdate` の改善でリストコンポーネントの更新処理自体は改善されています。 しかし、リストコンポーネントにリストアイテム（コメント）を追加する回数や頻度が多いと更新処理自体は改善されていても更新回数が増えます。

更新回数を減らすには、コメントを追加するタイミングを間引くような仕組みが必要です。 実はすでに、 `CommentThrottle` とそのままの名前のthrottlingで間引く処理の実装が使われていることがわかりました。

観測

このthrottlingの最適値を調べるつもりで、1秒ごとに5コのコメントを追加してその動きを調べてみました。

- 1秒ごとに5コのコメントを追加したとき
  - `onAddComments` は 4、1の引数で2回呼ばれる
- 1秒ごとに10コのコメントを追加した時
  - `onAddComments` は9、1の引数で2回呼ばれる
- 1秒ごとに2コのコメントを追加した時
  - `onAddComments` は 1、1 の引数で2回呼ばれる

なぜか2回に分けてコメントが追加されていることがわかりました。 一度に追加するコメント数を変えても `n -> 1` と2回に分けられています。

これは `CommentThrottle` の実装が何かおかしそうです。

修正の方針

意図した挙動は、Nミリ秒間にMコのコメントを追加したら、その後Mコのコメントが同時にリストコンポーネントに追加されるです。

まずは、その挙動になっているかテストするために `CommentThrottle` のテストコードを書くとよさそうです。

```
describe("CommentThrottle", () => {
    describe("コメントを一度も追加してない時", () => {
        it("intervalをまってもflushされない", () => {
            // テスト実装
        });
    });
    describe("コメントを追加した時", () => {
        it("追加しても同期的にはflushされない", () => {
            // テスト実装
        });
        it("追加したコメントはFLUSH_INTERVALまでflushされない", () => {
            // テスト実装
        });
        it("追加したコメントはFLUSH_INTERVAL後flushされる", () => {
            // テスト実装
        });
        it("複数のコメントを追加した場合は、FLUSH_INTERVAL後にまとめてflushされる", () => {
            // テスト実装
        });
        it("シナリオテスト", () => {
        });
    });
});
```

このテストを実装してみると、既存の `CommentThrottle` はバグがあることがわかりました。（意図した挙動のテストが通らない）

修正

まずは、テストが通るように無理やり `CommentThrottle` を修正しました。 その後、テストが通るのを維持したまま実装を書き換えて問題を修正しました。

fix(CommentThrottle): CommentThrottleの実装を修正 #7

Merged live merged 6 commits into `NicoliveComponent:master` from `live:fix-commentThrottle` on 13 Dec 2017

Conversation 10 Commits 6 Checks 0 Files changed 6

live commented on 8 Dec 2017 • edited Edited 1 time live edited on 11 Dec 2017 Member

概要

CommentThrottleのthrottle実装を修正。

詳細:

Before

コメントを複数追加した際にonAddCommentsが不思議な単位で呼ばれている

で次のようにコメントを定期的に投稿する。

post.sh

```
#!/bin/bash
lv=$2
comment_count=$1
count=1
while true
do
: ""${lv}"" "test ${count}" -t ${comment_count}
count=$((++count))
sleep 1
done
```

次は1秒ごとに5こコメントを投稿する

計測

テストが通るので大きな問題ないと思いますが、計測前に行った仕組みと同じ方法で呼び出す回数を記録しました。

- 1秒ごとに5コのコメントを追加したとき
  - `onAddComments` は 5の引数で1回呼ばれる
- 1秒ごとに10コのコメントを追加した時
  - `onAddComments` は10の引数で1回呼ばれる
- 1秒ごとに2コのコメントを追加した時 - `onAddComments` は 2の引数で1回呼ばれる

意図したとおりになっているので問題ありませんでした。

既存の実装が存在していても、その実装が意図したように動いているかは試してみないとわかりません。 実際に数値として出してみれば、実装が正しいかは分かるはず。

テストを書いたほうが結果的にコストが低い場合もあります。そのためテストを書いて試してみるのも大切です。

定期的に更新されているものを止める

映像を視聴する目的でページを閲覧していると、思っている以上に操作を行うことは少ないです。 そのため、何もしてないときに無駄な処理をしているものがいないかを見つけることは効果があるかもしれません。

ブラウザの多くの処理はイベント駆動なので、何もしてない状態で自動的に処理としてタイマーを使った処理があります。 ブラウザで定期的な処理を行う場合に使われるタイマーは次のものがあります。

- `setTimeout`
- `setInterval`
- `requestAnimationFrame`

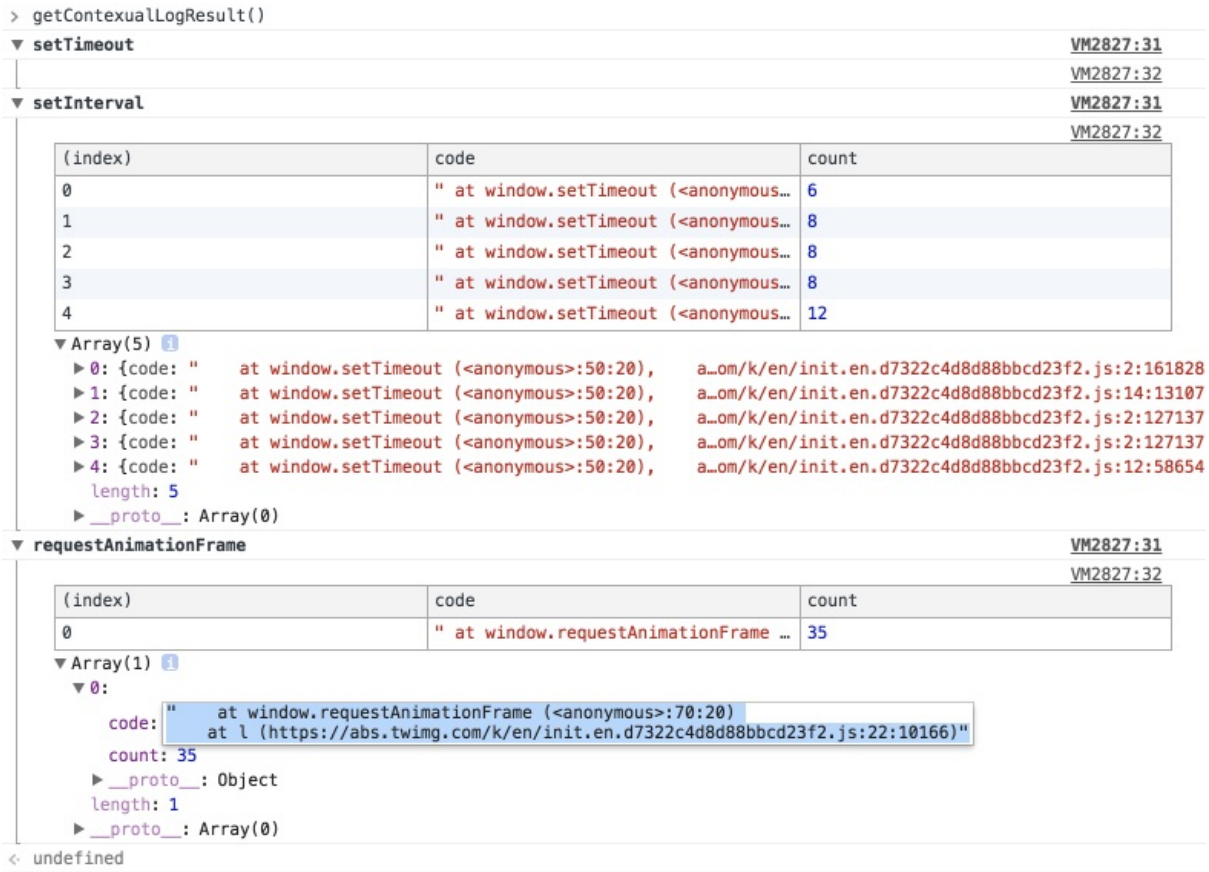
これらの関数を監視してみれば、無駄な処理を見つけることができそうです。

別の方法としてChromeにはPerformance Monitorがあるので、これをじっと見ていると変化に気づくことができます。 この問題も元々はPerformance Monitorを見ていて変な処理があることを見つけたのが出発点です。

- What's New In DevTools (Chrome 64) | Web | Google Developers

観測

次のようなスクリプトを書き、この状態で放置して動いてるタイマーを見つけます。




```
/**
 * ## Usage
 *
```

```
 * 1. Load following script
 * 2. `window.getContexualLogResult()` output the result to console
 *
 * ## Description
 *
 * - It spy "setTimeout", "setInterval", and "requestAnimationFrame".
 * - Collect call count and collect stack trace.
 */
// Disapprear log less than 5
const thresholdCount = 5;
const map = new Map([
    ["setTimeout", {}],
    ["setInterval", {}],
    ["requestAnimationFrame", {}]
]);

window.getContexualLogResult = () => {
    ["setTimeout", "setInterval", "requestAnimationFrame"].forEach(name => {
        const result = Object.entries(map.get(name))
            .filter(entry => {
                return entry[1] > thresholdCount;
            }).map(entry => {
                return {
                    code: entry[0],
                    count: entry[1]
                }
            });
        console.group(name);
        console.table(result);
        console.groupEnd(name);
    });
}
const contexualLog = (type, log) => {
    if (map.has(type)) {
        const object = map.get(type);
        const count = object[log] ? object[log] : 0;
        object[log] = count + 1;
    } else {
        const object = {};
        const count = object[log] ? object[log] : 0;
        object[log] = count + 1;
        map.set(type, object);
    }
}
```

定期的に動いてる無駄なものを停止する

```
48    const originalTimout = window.setTimeout;
49    window.setTimeout = function () {
50        const stack = (new Error().stack).split("\n").slice(1);
51        if (arguments.callee.caller) {
52            contexualLog("setTimeout", arguments.callee.caller.toString() + "\n" + stack);
53        } else {
54            contexualLog("setInterval", stack)
55        }
56        return originalTimout.apply(window, arguments);
57    }
58    const originalInterval = window.setInterval;
59    window.setInterval = function () {
60        const stack = (new Error().stack).split("\n").slice(1).join("\n");
61        if (arguments.callee.caller) {
62            contexualLog("setInterval", arguments.callee.caller.toString() + "\n" + stack);
63        } else {
64            contexualLog("setInterval", stack)
65        }
66        return originalInterval.apply(window, arguments);
67    }
68    const originalRaF = window.requestAnimationFrame;
69    window.requestAnimationFrame = function (callback) {
70        const stack = (new Error().stack).split("\n").slice(1).join("\n");
71        if (arguments.callee.caller) {
72            contexualLog("requestAnimationFrame", arguments.callee.caller.toString()
73        } else {
74            contexualLog("requestAnimationFrame", stack)
75        }
76        return originalRaF(callback);
77    }
```

timer-logging.js hosted with ❤ by GitHub view raw

参考

- ページ上でずっと動いているsetTimeout、setInterval、requestAnimationFrameを見つけてパフォーマンス改善する | Web Scratch

観測していると定期的に動いてるものとして、次のようなものがありました。

- Live Streamの映像取得
- コメントの取得
- コメントの描画
  - Canvasで描画されている
  - `requestAnimation` でメインループを回している
- 番組の状態の取得（視聴数やコメント数）
- アンケート画面の更新処理

- アンケートはゲームのようなメインループで描画を更新している。
- そのため `requestAnimation` でメインループを回している

この中で、アンケート画面が表示されていないにもかかわらず、アンケートのメインループ(`requestAnimation`)が動いていることがわかりました。これは無駄な処理といえそうです。

修正方針

アンケートが非表示にのときにもアンケートのメインループが動いているの問題だった。 原因としては、アンケートがない時は単にCSSで表示を消しているだけに過ぎない状態だった。

そのため、アンケートが行われていない時は、アンケートのプログラムを完全に消すようにすれば修正できそうです。

修正

アンケートが行われていない時は、アンケートをDOMからも削除するようにしました。

計測

観測で利用したスクリプトを使い、修正後に `requestAnimation` が減っていることが確認できました。

- ページ上でずっと動いているsetTimeout、setInterval、requestAnimationFrameを見つけてパフォーマンス改善する | Web Scratch

毎回関数を作ってpropsに設定する問題の修正

Reactのよくある問題として、propsに関数を渡すパターンで毎回関数を作って渡してしまうが知られています。 毎回新しい関数を作ってしまうと、同じ異なる値（props）となるため無駄にコンポーネントが更新されてしまいます。

why-did-you-updateでもこの問題を検知できます。

この問題は、関数同士の比較は常にtrueとしてしまう `shouldComponentUpdate` を実装するか、一度作った関数をちゃんと使いまわす（constructorで作成やpublic fieldを使う）ことで解決できます。

また、ESLintやTSLintといったLintツールでも検知できます。

- tslint: jsx-no-lambda
- eslint: jsx-no-bind

参考

- React.jsのrenderの戻り値の中で.bindで新しい関数を定義してはいけないわけ - Qiita

修正

視聴ページにもいくつかのコンポーネントで同じような問題がありました。 `tslint-react` を追加し、それらの問題を一個ずつ修正しました。

計測

why-did-you-updateを使って該当のコンポーネントの操作を行ったときに、無駄な更新ログがでないことが確認できました。

マウスを動かすと再描画する問題の修正

PCブラウザでもっと多く発生するイベントは`MouseMove`や`Scroll`であることが多いです。 そのため、これらのイベントを監視して更新処理をする時は慎重にならないといけません。

これらの頻度が高いイベントを軽減する方法としてdebounceや`Intersection Observer`などの新しいAPIがあります。

- Debouncing and Throttling Explained Through Examples | CSS-Tricks
- 入力ハンドラのデバウンス | Web | Google Developers

そもそもの問題として、そのイベントで何も変化する必要がないならばその更新処理を止めるべきです。

観測

視聴ページで、マウスを移動をすると載せただけで更新されるコンポーネントが幾つかあることがわかりました。

facebook/react-devtoolsの"Highlight Updates"を使うことで更新されているコンポーネントが点滅します。

マウス移動で何か表示が変わっているのなら、それは意図した挙動なので問題ありません。 しかし、表示が変わっていないにもかかわらず更新処理（点滅）を行っているならばそれは無駄といえます。

why-did-you-updateを使うことで、無駄な更新処理なのかを確認してみました。 次のような感じでReactをラップすると、無駄な更新をコンソールに表示してくれます。

```
import * as React from "react";
```

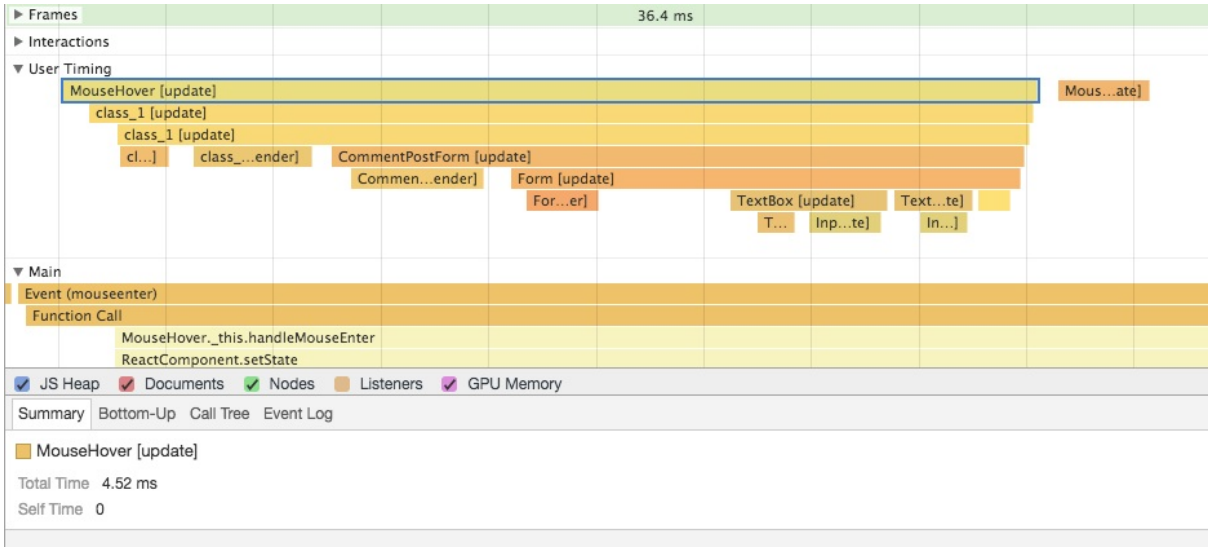
```
if (process.env.NODE_ENV !== "production") {
    const { whyDidYouUpdate } = require("why-did-you-update");
    whyDidYouUpdate(React);
}
```

ここでの"無駄な更新"とは、全く同じ値（Props）なのに更新(`render`)されている処理のことを言います。

- why-did-you-update
  - 無駄な更新をコンソールに出力してくれる
  - たとえばあるアクションにおいて無駄なレンダリングが起きてるかを調べるときに使う
- react-addons-perf（React 15限定）
  - “Wasted” time がwhy-did-you-updateと似た感じ
  - 一覧できるので分かりやすい
  - アプリを起動時や色んな操作をした結果の統計を見るのに使う
- react-performance
  - React 16対応
  - 開発中

why-did-you-updateを入れた状態でマウス移動を行う特定のコンポーネントがコンソールログに表示されました。

合わせてコメント欄周辺をマウス移動した時のプロファイルを取ってみます。



先ほどのwhy-did-you-updateとの出力と合わせてみると、`Form`コンポーネントの下にある、`Input`や`Button`や`Field`といったコンポーネントが無駄に更新されていることがわかります。

問題

次のコンポーネントが同じPropsを受け取ったにもかかわらず更新されているのが問題です。

- `Input`
- `Button`
- `Field`

実装を見ると`React.Component`を継承していて、かつ`shouldComponentUpdate`が実装されていませんでした。

```
export class Field extends React.Component {}
```

この場合はPropsが変わるたびに更新されます。

修正方針

`shouldComponentUpdate`を実装して、前回と同じPropsなら更新しないようにすれば解決できそうです。

Note

この問題の修正のために独自の`DeepComparisonRenderingOptimizedComponent`という親コンポーネントを使っています。（既存の構造の問題があったため） 通常のケースではReactが提供している`React.PureComponent`のShallow（浅い）な比較で十分でしょう。

Shallowな比較と、Deepな比較の違いについては次の記事で解説しています。

- shallow-equal for Object/React props | Web Scratch

修正

該当するそれぞれのコンポーネントに`shouldComponentUpdate`を実装を追加しました。

pref: PureRenderingOptimizedComponentの追加 #1108

Merged live merged 10 commits into NicoliveComponent:v6 from live:pure-component-views on 15 Dec 2017

Conversation 11 Commits 10 Checks 0 Files changed 11

live commented on 11 Dec 2017 • edited Edited 1 time live edited on 15 Dec 2017 Member

概要

マウス移動で無駄な更新が走っているコンポーネントの修正

問題点

- マウスを動かすたびに、更新する必要のないコンポーネントが更新されている
- PlayerControllerとCommentPostPannelのなかにある要素が更新されていた
  i. MouseHover が自分へ `setState({ mouseHovering: true })` する
  ii. MouseHoverの子(PlayerControllerやCommentPostPannel)には `mouseHovering` がpropsで渡される
  iii. PlayerControllerより下の子要素は `mouseHovering` を使ってない
  iv. そのため子要素は更新する必要がないが、更新されていた

無駄な更新(値が同じにもかかわらずrenderされる)はwhy-did-you-updateを使うことで検出した。
無駄な更新がコンソールログに出力される。

`LeoPlayerViewComponent` の Storyに追加した状態で計測した。

改善方法

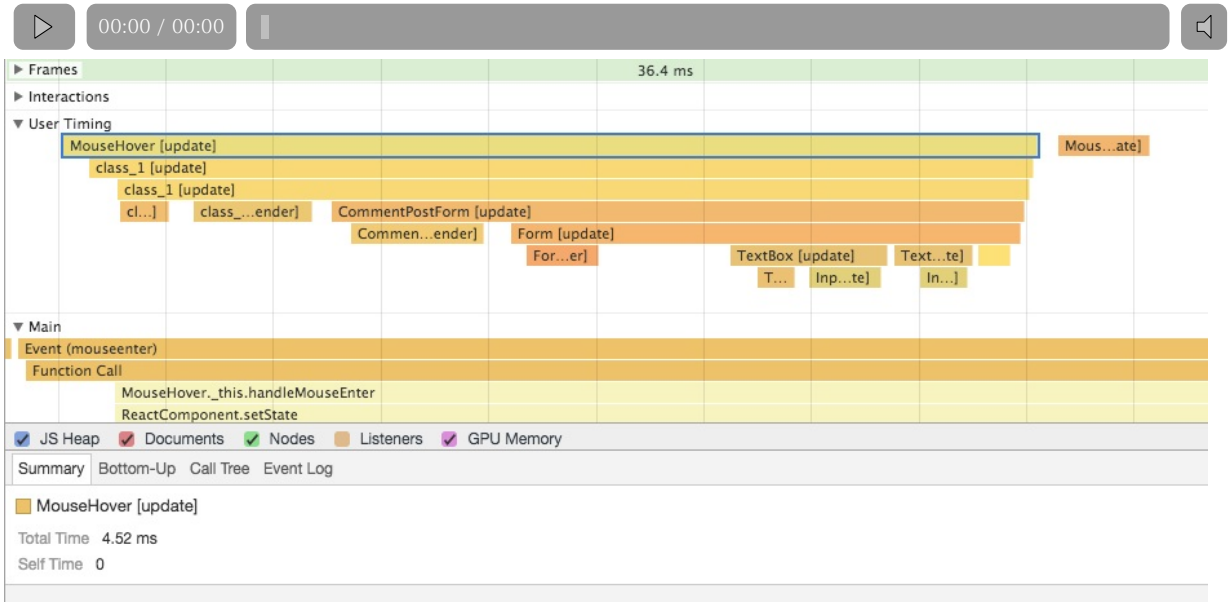
3行

- Shallow EqualするPureComponentでは更新がとまらない
- Deep Equalする `DeepComparisonRenderingOptimizedComponent` を追加して更新をとめる
- propsのクローンをやめればPureComponentでもよくなるはず

計測

Before

マウスを動かすとwhy-did-you-updateのログがでています。



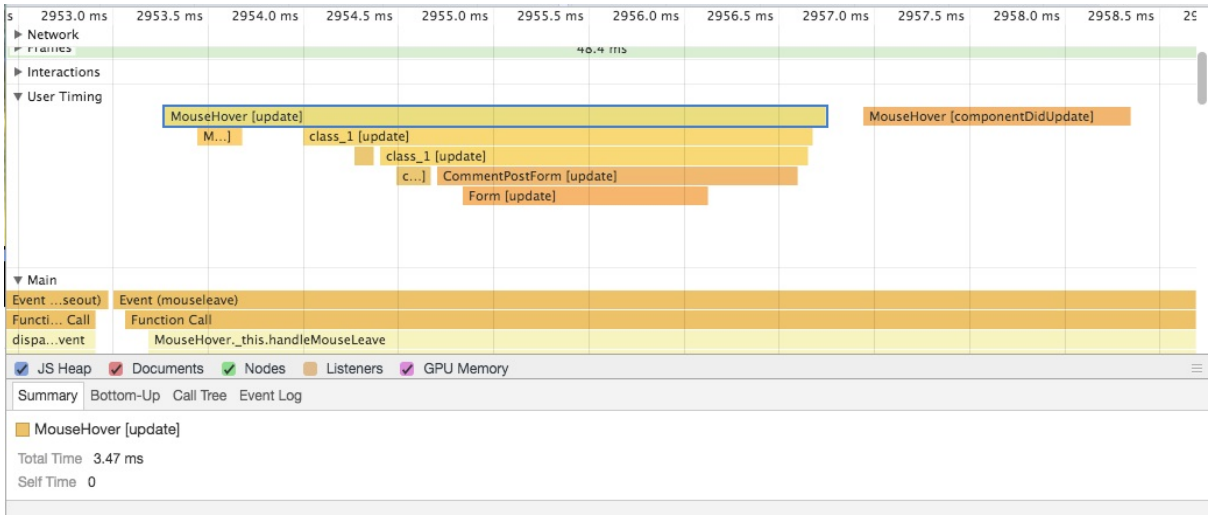
After

マウスを動かしても無駄な更新されるコンポーネントはなくなりました。 why-did-you-updateのログが出なくなった)

00:00 / 00:00

具体的には、`Form` まで `update` 処理が止まっていることがプロファイルからわかります。 つまり、`Form` 次のコンポーネントにおいて `shouldComponentUpdate` により更新が不要と判断され、`update` 処理が行われなくなっていることを意味します。

- `Input`
- `Button`
- `Field`



トレードオフの検証

紹介した修正例では既存の挙動は維持したまま、目には見えない改善をしていました。 現実的にはそのアプリケーション自体の機能を減らさないと、低スペックのデバイスではそもそも無理な場合があります。

そのような機能の削減とパフォーマンスのトレードオフを検証する必要があります。

生放送の視聴ページではこのトレードオフを"軽量モード"というスタンスで進めることにしました。 軽量モードとは最終的にユーザが設定で選択して初めて機能を落とす、オプトイン方式の設定です。

軽量モード

色々な機能を削ってみて、何が効果的なのかを検証する必要があります。 機能を更新するたびステージングサイトにあげて検証というのは大変なので、簡単なURLで切り替えるできる仕組みを追加して検証しました。

実際に使った実装はものすごく単純です。 productionビルドでは必ず無効にするguardを入れておき、後はURLのクエリで切り替えできるという仕組みです。

```
const enableLightMode = process.env.NODE_ENV !== "production" && /[?&]leo_light_mode\b/.test(location.href);
// まとめて有効化
// &leo_light_mode=comment-panel-hidden,comment-panel-fixed-bottom,comment-fps,comment-resolution
// コメントパネルの非表示化
// &leo_light_mode=comment-panel-hidden
export const isHiddenCommentPanel = enableLightMode
    && /[?&]leo_light_mode=.*comment-panel-hidden\b/.test(location.href);
// コメントパネルの最下部への追従を停止
// &leo_light_mode=comment-panel-fixed-bottom
export const isDisabledCommentPanelFixedBottom = enableLightMode
    && /[?&]leo_light_mode=.*comment-panel-fixed-bottom\b/.test(location.href);
// コメントパネルのバッファサイズを指定
// &leo_light_mode=comment-panel-buffer-size:0
export const commentPanelBufferSizeMatch = enableLightMode
    && location.href.match(/[?&]leo_light_mode=.*comment-panel-buffer-size:([\d.]+)\b/);
export const commentPanelBufferSize = commentPanelBufferSizeMatch !== false && commentPanelBufferSizeMatch !==
null
    ? parseFloat(commentPanelBufferSizeMatch[1])
    : undefined;
// 映像要素を非表示化
// &leo_light_mode=video-hidden
export const isHiddenEdgeStream = enableLightMode
    && /[?&]leo_light_mode=.*video-hidden\b/.test(location.href);
// コメントレンダラーのWebGLの有効化
// &leo_light_mode=comment-render-webgl
export const useWebGLCommentRenderer = enableLightMode
    && /[?&]leo_light_mode=.*comment-render-webgl\b/.test(location.href);
if (enableLightMode) {
    console.log(`
# 軽量モードのフラグ状態
コメントパネルの非表示化: ${isHiddenCommentPanel}
コメントパネルの最下部への追従を停止: ${isDisabledCommentPanelFixedBottom}
コメントパネルのバッファサイズ: ${commentPanelBufferSize}
映像要素を非表示化: ${isHiddenEdgeStream}
コメント描画のWebGL有効化: ${useWebGLCommentRenderer}
`);
}
```

その軽量モードのドキュメントは次のような単純なものでした。

```
## 有効化の方法
```

```
URLに `&leo_light_mode=comment-panel-hidden,comment-panel-fixed-bottom,comment-fps,comment-resolution` などそれぞれの機能を無効化できるフラグを設定できる。複数渡す時は `,` など適当なもので区切る。

## 軽量モードの種類

- コメントパネルの非表示化
  - `&leo_light_mode=comment-panel-hidden`
- コメントパネルの- 最下部への追従を停止
  - `&leo_light_mode=comment-panel-fixed-bottom`
- コメントパネルのバッファサイズを指定
  - `&leo_light_mode=comment-panel-buffer-size:バッファサイズ`
  - 例) バッファサイズを`0`にする
  - `&leo_light_mode=comment-panel-buffer-size:0`
- 映像要素を非表示化
  - `&leo_light_mode=video-hidden`
- コメントレンダラーのWebGLの有効化
  - `&leo_light_mode=comment-render-webgl`
```

検証

後は、実際にURLにクエリをつけて何を外したら何がよくなるかというトレードオフを地道に検証していくだけです。

その後、"軽量モード"をオプションとして本当に実装するときにフラグを取り除けばよいだけです。

ライブラリを改善する

ニコニコ生放送（PC）では生放送の映像を再生するためhls.jsを利用しています。 hls.jsはHLSに非対応ブラウザ（ChromeやFirefoxなど）でもHLSを再生するためのライブラリです。

HLS自体については次のサイトなどを見てください。

- HTTP Live Streaming (HLS) - Apple Developer
- HLSについて知っていることを話します
- フロントエンドエンジニアのための動画ストリーミング技術基礎 | ygoto3.com

指標についてでも書いていますが、視聴ページのメインコンテンツは映像、映像の上に描画するコメントです。

`hls.js` はこの映像を再生に利用するため、 `hls.js` のパフォーマンスはそのまま視聴ページのパフォーマンスや安定性に繋がります。

ここではまず `hls.js` のパフォーマンスを分析し、改善できる点がないかを探すことにしました。

関数プロファイル

hls.jsはMedia Source Extensions（MSE）というAPIを利用し映像を再生します。 `hls.js` ではこのMSEに渡せるように、映像を取得、デコードまた映像のバッファをチェックするといった処理が行われています。

`hls.js` の処理はDOM APIにほとんど依存してない（一部Web Workerなどがあります）処理なので、そのままJavaScriptコードのプロファイルを取ればボトルネックが見えてきそうです。

ブラウザの開発者ツールでもプロファイルを取れますが、開発者ツールだと他のコードなどもまざってしまいノイズが多くなってしまいました。 そこで、node-sjspを少し改変して `hls.js` のコードだけのプロファイルを取れるようにしたものを使いました。

- 45deg/node-sjsp: sjsp (Simple JavaScript Profiler) implemented in Node.js

コードも一部改変しましたが、次のような `node-sjsp` を使ってコード変換できるwebpackのLoaderを書きました。このLoaderを特定のパス(`hls.js`)のパスだけに適応すれば、特定のコードだけの関数プロファイルがコンソールに出力されます。

```
"use strict";
const inject = require("node-sjsp").inject;
const path = require("path");
const currentDir = __dirname
module.exports = function (jsCode, inMap) {
    var filepath = path.relative(currentDir, this.resourcePath);
    var injectedCode = inject(filepath, jsCode, 10);
    return injectedCode;
};
```

実際に取得した `hls.js` の関数ごとのプロファイルを見てみてみます。

(index)	time ▼	count	name	f…	line
0	434	1988	"observer.trigger"	"…	" observer.trigger = function trigger(event) {"
1	422	1988	"EventEmitter.prototype.emit"	"…	"EventEmitter.prototype.emit = function(type) {"
2	384	2529	"EventHandler.prototype.onEvent"	"…	" EventHandler.prototype.onEvent = function onEvent(event, data) {"
3	377	2529	"EventHandler.prototype.onEventGeneric"	"…	" EventHandler.prototype.onEventGeneric = function onEventGeneric(event, data) {"
4	291	672	"StreamController.prototype.tick"	"…	" StreamController.prototype.tick = function tick() {"
5	284	672	"StreamController.prototype.doTick"	"…	" StreamController.prototype.doTick = function doTick() {"
6	204	481	"StreamController.prototype._doTickIdle"	"…	" StreamController.prototype._doTickIdle = function _doTickIdle() {"
7	146	363	"XhrLoader.prototype.readystatechange"	"…	" XhrLoader.prototype.readystatechange = function readystatechange(event) {"
8	142	324	"BufferController.prototype.onSBUpdateEnd"	"…	" BufferController.prototype.onSBUpdateEnd = function onSBUpdateEnd() {"
9	107	324	"StreamController.prototype.onBufferAppended"	"…	" StreamController.prototype.onBufferAppended = function onBufferAppended(data) {"
10	105	404	"StreamController.prototype._checkAppendedParsed"	"…	" StreamController.prototype._checkAppendedParsed = function _checkAppendedParsed() …
11	92	41	"PlaylistLoader.prototype.loadsuccess"	"…	" PlaylistLoader.prototype.loadsuccess = function loadsuccess(response, stats, conte…
12	89	480	"get"	"…	" get: function get() {"
13	85	480	"get"	"…	" get: function get() {"
14	84	480	"get"	"…	" get: function get() {"
15	79	479	"StreamController.prototype._fetchPayloadOrEos"	"…	" StreamController.prototype._fetchPayloadOrEos = function _fetchPayloadOrEos(pos, b…
16	75	480	"get"	"…	" get: function get() {"
17	67	479	"get"	"…	" get: function get() {"
18	65	80	"StreamController.prototype._loadFragmentOrKey"	"…	" StreamController.prototype._loadFragmentOrKey = function _loadFragmentOrKey(frag, …
19	61	244	"Demuxer.prototype.onWorkerMessage"	"…	" Demuxer.prototype.onWorkerMessage = function onWorkerMessage(ev) {"
20	61	40	"StreamController.prototype.onLevelLoaded"	"…	" StreamController.prototype.onLevelLoaded = function onLevelLoaded(data) {"
21	60	121	"XhrLoader.prototype.load"	"…	" XhrLoader.prototype.load = function load(context, config, callbacks) {"
22	58	121	"XhrLoader.prototype.loadInternal"	"…	" XhrLoader.prototype.loadInternal = function loadInternal() {"
23	46	1621	"bufferInfo"	"…	" bufferInfo: function bufferInfo(media, pos, maxHoleDuration) {"
24	45	80	"FragmentLoader.prototype.onFragLoading"	"…	" FragmentLoader.prototype.onFragLoading = function onFragLoading(data) {"
25	40	160	"StreamController.prototype.onFragParsingData"	"…	" StreamController.prototype.onFragParsingData = function onFragParsingData(data) {"
26	32	672	"StreamController.prototype._checkBuffer"	"…	" StreamController.prototype._checkBuffer = function _checkBuffer() {"
27	31	80	"FragmentLoader.prototype.loadsuccess"	"…	" FragmentLoader.prototype.loadsuccess = function loadsuccess(response, stats, conte…
28	29	672	"StreamController.prototype._checkFragmentChanged"	"…	" StreamController.prototype._checkFragmentChanged = function _checkFragmentChanged(…
29	29	38	"LevelController.prototype.loadLevel"	"…	" LevelController.prototype.loadLevel = function loadLevel() {"

▶ Array(30)

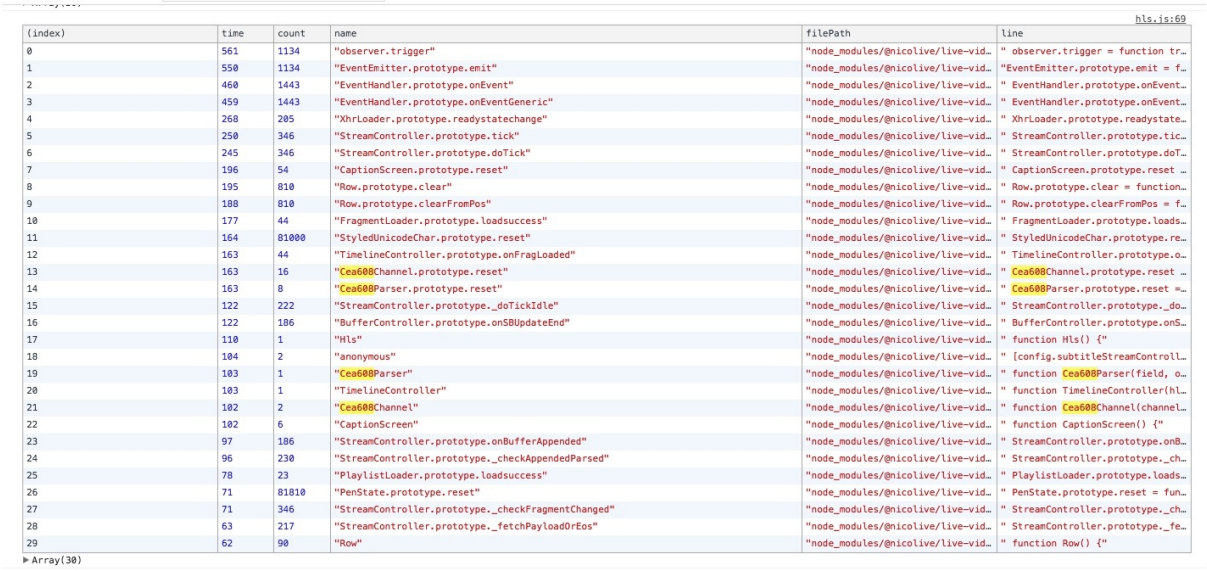
合計の処理時間順に並べてるとイベントの仕組みやメインループでの `tick` 、バッファチェックなどが大部分を占めていることがわかります。

- `EventEmitter`
- `tick` （バッファのチェックなどをして正しく再生できているかを100msごとにチェックする）
- `_checkAppendedParsed` （バッファが断片化してないかをチェックする）

それぞれの `tick` 1回の処理は数ミリ秒などとても小さいですが、 `hls.js` の仕組み上どこかで数百ミリ秒の処理が発生すると、それは映像がそこで止まるということを意味します。 そのため、常に安定した時間で処理を回し続けることが大切であるという認識です。

不要な処理を無効化

他にも `node-sjsp` を使っていろいろな状況でプロファイルを取っていると、 `Cea608` という見慣れない処理が出てくることがありました。



hls.js:69

(index)	time	count	name	filePath	line
0	561	1134	"observer.trigger"	"node_modules/@nicolive/live-vid…	" observer.trigger = function tr…
1	550	1134	"EventEmitter.prototype.emit"	"node_modules/@nicolive/live-vid…	"EventEmitter.prototype.emit = f…
2	460	1443	"EventHandler.prototype.onEvent"	"node_modules/@nicolive/live-vid…	" EventHandler.prototype.onEvent…
3	459	1443	"EventHandler.prototype.onEventGeneric"	"node_modules/@nicolive/live-vid…	" EventHandler.prototype.onEvent…
4	268	205	"XhrLoader.prototype.readystatechange"	"node_modules/@nicolive/live-vid…	" XhrLoader.prototype.readystate…
5	250	346	"StreamController.prototype.tick"	"node_modules/@nicolive/live-vid…	" StreamController.prototype.tic…
6	245	346	"StreamController.prototype.doTick"	"node_modules/@nicolive/live-vid…	" StreamController.prototype.doT…
7	196	54	"CaptionScreen.prototype.reset"	"node_modules/@nicolive/live-vid…	" CaptionScreen.prototype.reset …
8	195	810	"Row.prototype.clear"	"node_modules/@nicolive/live-vid…	" Row.prototype.clear = function…
9	188	810	"Row.prototype.clearFromPos"	"node_modules/@nicolive/live-vid…	" Row.prototype.clearFromPos = f…
10	177	44	"FragmentLoader.prototype.loadsuccess"	"node_modules/@nicolive/live-vid…	" FragmentLoader.prototype.loads…
11	164	81000	"StyledUnicodeChar.prototype.reset"	"node_modules/@nicolive/live-vid…	" StyledUnicodeChar.prototype.re…
12	163	44	"TimelineController.prototype.onFragLoaded"	"node_modules/@nicolive/live-vid…	" TimelineController.prototype.o…
13	163	16	"Cea608Channel.prototype.reset"	"node_modules/@nicolive/live-vid…	" Cea608Channel.prototype.reset …
14	163	8	"Cea608Parser.prototype.reset"	"node_modules/@nicolive/live-vid…	" Cea608Parser.prototype.reset =…
15	122	222	"StreamController.prototype._doTickIdle"	"node_modules/@nicolive/live-vid…	" StreamController.prototype._do…
16	122	186	"BufferController.prototype.onSBUpdateEnd"	"node_modules/@nicolive/live-vid…	" BufferController.prototype.onS…
17	110	1	"Hls"	"node_modules/@nicolive/live-vid…	" function Hls() {"
18	104	2	"anonymous"	"node_modules/@nicolive/live-vid…	" [config.subtitleStreamControll…
19	103	1	"Cea608Parser"	"node_modules/@nicolive/live-vid…	" function Cea608Parser(field, o…
20	103	1	"TimelineController"	"node_modules/@nicolive/live-vid…	" function TimelineController(hl…
21	102	2	"Cea608Channel"	"node_modules/@nicolive/live-vid…	" function Cea608Channel(channel…
22	102	6	"CaptionScreen"	"node_modules/@nicolive/live-vid…	" function CaptionScreen() {"
23	97	186	"StreamController.prototype.onBufferAppended"	"node_modules/@nicolive/live-vid…	" StreamController.prototype.onB…
24	96	230	"StreamController.prototype._checkAppendedParsed"	"node_modules/@nicolive/live-vid…	" StreamController.prototype._ch…
25	78	23	"PlaylistLoader.prototype.loadsuccess"	"node_modules/@nicolive/live-vid…	" PlaylistLoader.prototype.loads…
26	71	81810	"PenState.prototype.reset"	"node_modules/@nicolive/live-vid…	" PenState.prototype.reset = fun…
27	71	346	"StreamController.prototype._checkFragmentChanged"	"node_modules/@nicolive/live-vid…	" StreamController.prototype._ch…
28	63	217	"StreamController.prototype._fetchPayloadOrEos"	"node_modules/@nicolive/live-vid…	" StreamController.prototype._fe…
29	62	90	"Row"	"node_modules/@nicolive/live-vid…	" function Row() {"

▶ Array(30)

CEA-608はキャプション（字幕）のことです。 字幕がない動画に対してもこの処理が行われているため、なにか無駄な処理を行っていそうです。

`hls.js` のソースコードやドキュメントを見るとenableCEA708Captionsというオプションで字幕処理を無効化できることがわかります。 （これらはデフォルトが有効です）

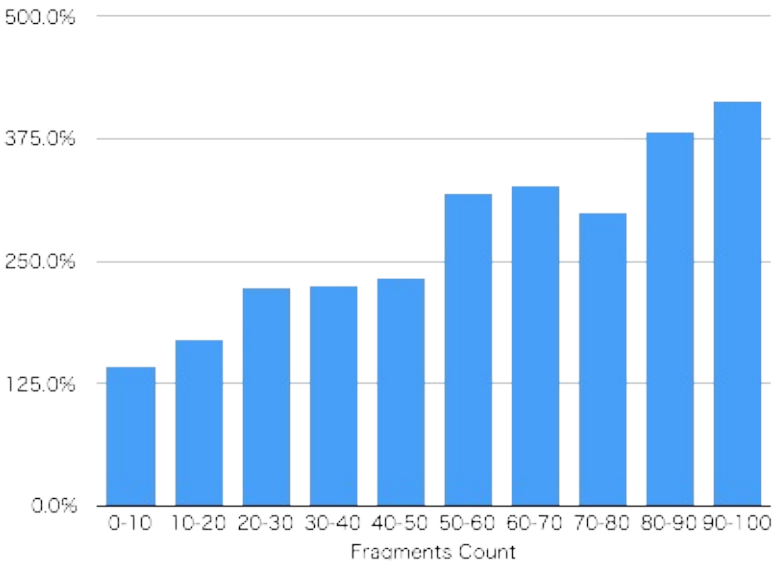
このenableCEA708Captionsオプションを `false` にすることで `Cea608` に関する処理をしなくなることが確認できました。

hls.jsではこのオプションが有効時にフラグメントを取得するたびに、 `cea608Parser` をリセットする処理が行われていました。

- https://github.com/video-dev/hls.js/blob/905e65fd68095bd5e2bc9f888ae09b64fca8835a/src/controller/timeline-controller.js#L228-L232

`hls.js` を修正する

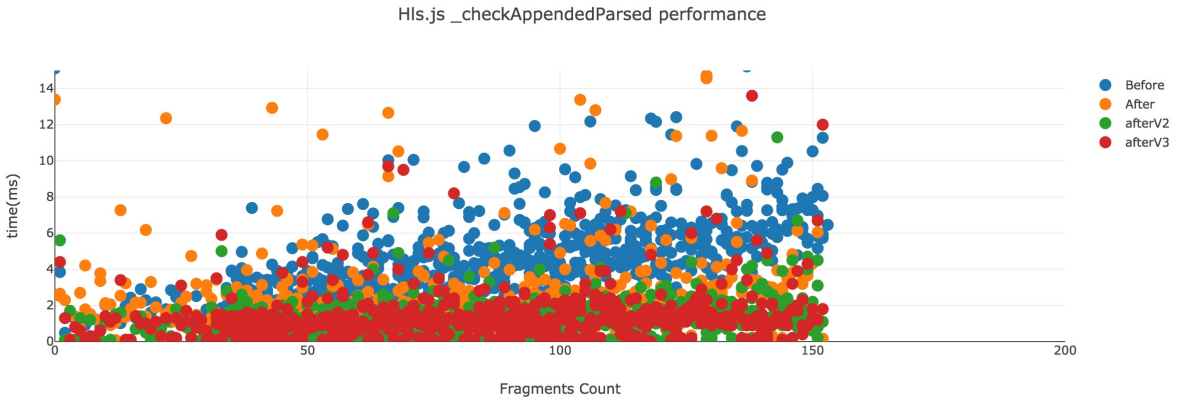
関数のプロファイルを見てみると `hls.js` で処理の中心となっているのは高頻度が呼ばれる `tick` や `_checkAppendedParsed` などでした。 これらの処理は高頻度で呼ばれるため、少しでも改善すると映像再生の安定化に寄与するだろうと仮説がありました。 また、 `_checkAppendedParsed` などは保持しているフラグメントの数（HLSは映像がセグメント単位に細かく切ったものを取得し再生します）と処理時間に相関がありそうでした。 実際に `fragments` という配列に対して、forループでチェック処理を行うため、フラグメントの数が増えると処理時間が増加していました。



> フラグメント数と`_checkAppendedParsed`の処理コストの相関

これらのパフォーマンスを改善するには`hls.js`そのものに手を入れる必要があるため、`hls.js`にPull Requestして修正するようにしました。

さきほどの`_checkAppendedParsed`の問題もループ内の処理を最適化するPRを出すなどして、マイクロベンチ上は3.38倍早くなりました。



> Improve `StreamController#_checkAppendedParsed` performance #1528

これに加えてバッファチェックの仕組みの構造を修正するといった改善も行っています。

また`hls.js`は複雑なステートをもつコードですが、それに対してテストの量が足りていなかったのでテストを追加したり、 SauceLabを使ったCIが不安定だったのを問題を安定化するなどを行いました。

最終的には数十のコミットやPull Requestをした結果、`hls.js`のコラボレータとして活動しています。

ForkではなくPull Request

hls.jsのパフォーマンス改善を行う場合に、Pull Requestを送るのではなくForkしてしまうという選択肢もありました。 しかしながら、`hls.js`は十分複雑なライブラリです。そのため安易にForkすると、Upstreamに追従するのも難しくメンテナンスコストが高くなりやすいです。

- Living Downstream - BOF vFinal.key

- SonyがLLVMのForkをやめた話

これは `hls.js` に限らずさまざまなOSSを使った開発で発生する問題です。 あるライブラリを利用する際には、そのライブラリに問題が発生したときにどうするかを考えてから選択しても遅くはないかもしれません。

ページ表示速度の改善

視聴中のパフォーマンス改善（ランタイム）が一段落し、次はページロードを改善する作業にとりかかろうとしました。

ランタイムのパフォーマンス計測に比べて、ページロードのパフォーマンス計測は、計測手法や改善方法がある程度体系化されています。 一方でページロードの時間を継続的に改善する/維持するには仕組みが必要です。

また今回のパフォーマンス改善も"目的"を決めてから改善に取りかかりました。 今回のページロードのパフォーマンス改善において決めた目的は次のとおりです。

目的

- 初期表示における表示速度を改善する
- 動画部分の表示がより高速に表示できることを目的とする

このような目的を立て現状の調査から行うことにしました。 その調査の中でJavaScriptとCSSどちらもにファイルサイズの問題を抱えていることがわかりました。 まずは初期表示に必要なファイルサイズを小さくするという点に絞って話をすすめていくことにしました。

また、これらのページ表示速度の改善といったパフォーマンス改善には基本的に終わりはありません。 そのため、改善中に何をしたかや何をしてないかを一覧できるMeta的なIssueを作成し、そこにIssueやPRへのリンクを集めるようにしました。 GitHubは自動的にクロスリファレンスとなるため、リンクを貼るだけでなにが行われたが一覧しやすくなります。

実際に改善作業を行う前に、まずそのファイルサイズの変化を計測する仕組みが必要です。 WebPagetestを使って継続的にファイルサイズなどのパフォーマンスデータを集める仕組みを作ることからはじめました。

- 継続的なパフォーマンス計測

その後、実際にさまざまなアプローチでファイルサイズを減らしていきました。

- パッケージはBundleを配布しない
- "module"フィールド対応
- ファイルサイズを減らす

参考

最初に述べたように、視聴中のパフォーマンス改善（ランタイム）に比べて、ページロードのパフォーマンス改善はすでにさまざまな書籍や文書があります。 この文書もその中の一部に過ぎません。他の文書も参考にしてください。

- thedaviddias/Front-End-Performance-Checklist: The only Front-End Performance Checklist that runs faster than the others
- 超速！ Webページ速度改善ガイド ——使いやすさは「速さ」から始まる：書籍案内｜技術評論社
- Using WebPageTest - O'Reilly Media
- High Performance Browser Networking (O'Reilly)
- Webフロントエンド ハイパフォーマンス チューニング｜技術評論社
- Webpagetestから始める継続的パフォーマンス改善

計測

ページロードのパフォーマンスを計測については、すでに多くのツールやサービスが存在します。

以下はパフォーマンス計測を行えるサービスの例です。

- WebPagetest（無料）
- SpeedCurve（有料）
- Calibre（有料）
- New Relic Synthetics（有料）
- CatchPoint（有料）

今回の対象のサイトはログインが必須であったことと、とりあえずないよりはあったほうがいいという考えであったため、 手軽に使えるWebPagetestで計測を行うことにしました。 （環境によって左右されにくいファイルサイズをメインとしてのも理由の1つです）

WebPagetest

WebPagetestはOSSとして公開されていてプライベートインスタンスを立てることもできます。 また、ホスティング版では制限はありますが、API経由でパフォーマンス計測を行いその結果を取得できます。 しかし、他のサービスのように結果を連続して見られるダッシュボードのような機能はありません。

Sitespeed.ioのようなWebPagetestと連携できるパフォーマンスモニタリングツールもあります。

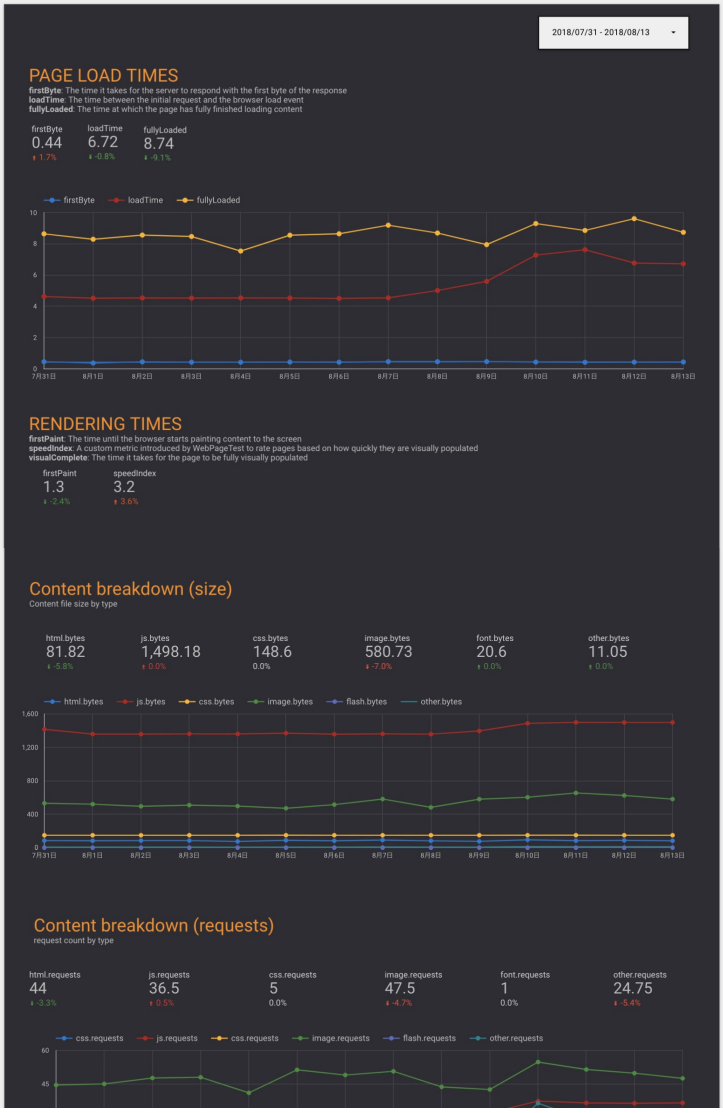
今回は別途サーバなどを用意せずに簡単に導入できるGoogle Apps Scriptを使い、Google Spreadsheetに結果を記録し、Google DataStudioで結果を見るダッシュボードを作るという方法を取りました。

実際に利用したものは次にリポジトリに公開されているGoogle Apps Scriptです。 これを利用することで30分に1回程度計測を行い、その結果をSpreadSheetに記録して、Google DataStudioで見られる環境を作成しました。

Googleアカウントのみで完結して、Cron的な仕組みもGoogle Apps Scriptで行えるので値を記録するだけならシンプルです。

- uknmr/gas-webpagetest
- DataStudioとGASでWebPagetestの計測結果をグラフ化する | mediba Creator × Engineer Blog

Google DataStudioを使うことでSpreadSheetなどのデータ元にそれを可視化するダッシュボードを作成できます。これでWebPagetestで特定のページを計測して、そのページのロード時間やコンテンツ（HTML, JS, CSSなど）のサイズを継続的に監視できるようになりました。



経過を監視する

これらの方法でパフォーマンス計測を行うことには主に2つの意味があります。

1. パフォーマンス改善を行い、その結果を外部から観測して確認する
2. パフォーマンス改善以外の変更で、パフォーマンスが落ちてないかを検知する

どちらもパフォーマンスに関する値を継続的に取ることで、その連続性から変化に気づくことができます。

ファイルサイズなどの値としてはっきりしているものならば、修正を反映するたびにChrome DevToolsなどで確認することもできますが、手動で行うには大変です。 また、パフォーマンスに関係しない機能の追加などをした際に思わぬところで、パフォーマンスへ悪影響を与える場合があります。 そのような問題に気づくためにも、継続的にパフォーマンスを計測することは必要です。

参考

- Webpagetestから始める継続的パフォーマンス改善

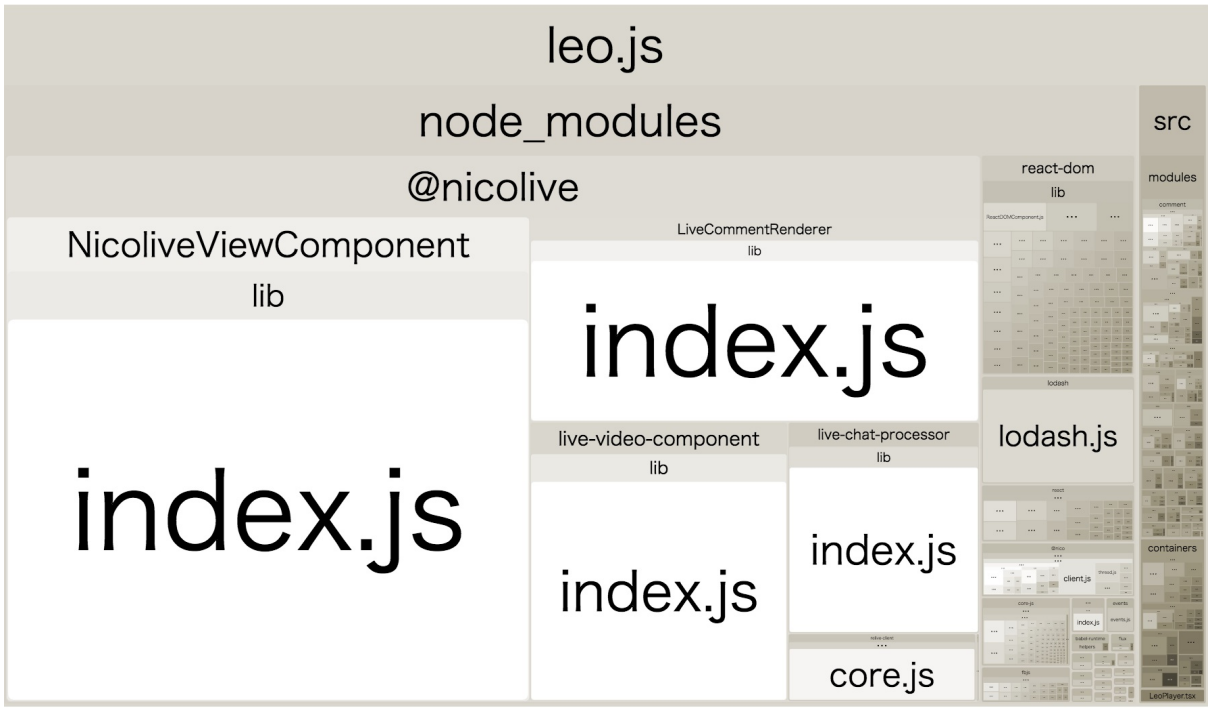
パッケージはBundleを配布しない

ニコニコ生放送 PCのHTML5プレイヤーは、基本的にはTypeScriptで書かれていてビルドにはwebpack + tsloaderという構造になっています。

webpackでビルドされているのでWebpack Bundle Analyzerを使い、ビルドしたパッケージの依存をビジュアライズして調査しました。

- Webpack Bundle Analyzer

実際の改善前のbundleしたファイル構造は次のような形でした。



`@nicolive` から始まるものは社内のnpmレジストリに公開されている社内モジュールです。 この社内モジュールが `lib/index.js` という大きな1つファイルしか持っていないことがわかります。

これは、各パッケージがbundle済みのファイルを `lib/index.js` として配布するようになっていたためです。また、そのbundle済みファイルを `package.json` の `main` に指定していたため、単純に `require` などをするとbundle済みファイルがよみこまれるようになっていました

Bundle済みのファイルだけを配布する問題

パッケージがbundle済みのファイルだけを配布するメリットとデメリットは次のようになります。

メリット

- ライブラリ側にビルドしないと使えない特殊な仕組みがある場合に、使う側が何も考えずに利用できる
  - 例) hls.jsは内部的にWeb Workerを使うためbundle済みファイルを配布をしている
- Direct require(`require("module/lib")`)で非公開なファイルを参照されるのを防げる
  - React v16.0ではRollupで最適化したbundle済みファイルを配布することで `react/lib/*` への参照を防いでいます

デメリット

- ライブラリ側が依存しているモジュールもbundleに含まれてしまう
  - たとえば、ライブラリが `lodash` を使っている場合に、bundleにも `lodash` が含まれてします
  - ライブラリを使う側も `lodash` を使っている場合は、重複して `lodash` が入ってしまう
- ライブラリを使う側でより積極的な最適化が行えない
  - bundle済みのファイルを配布するということは、ES5へ変換済みのコードを配布するということになる
  - Tree Shakingのようなアプリとライブラリを含めた全体で最適化するビルドが適応できなくなる

実際には、ライブラリはbundle済みのファイルとTypeScriptをJavaScriptに変換しただけのファイルの2種類を同時に配布できます。

今回の調査でライブラリ `lodash` が使われていたものが複数あり、 その結果、複数の `lodash` がアプリに含まれていました(`lodash` は1つ20kb gzip)。

この問題を解消するために、ライブラリはそれぞれ二種類のファイルを出力するように修正しました。

- bundle済みのファイルを `dist/` へ
- tscで変換したファイルを `lib/` へ

Before:

```
├── lib/
|      bundleしたファイル
├── src/
|      TypeScriptのソースコード
```

After:

```
├── dist/
|      bundleしたファイル
├── lib/
|      src/をJavaScriptに変換したファイル
├── src/
|      TypeScriptのソースコード
```

これによって、ライブラリの利用者側でbundle済みか `lib/` したのファイルを使うが選択できるようになりました。

- 開発時は `src/` 次のソースコードを編集する
- ライブラリとして使う場合はJavaScriptに変換済みの `lib/` が参照される
- scriptタグなどで読み込む場合は `dist/` のUMDへbundle済みファイルを直接利用する

この作業を社内に公開してるモジュールにそれぞれ適応しました。

計測

それぞれのライブラリがbundle済みではなくなったので、各ライブラリで重複したモジュールがなくなりました。
最終的にすべてをbundleしたJavaScriptのファイルサイズが小さくなることが確認できました。

- 2.7Mb -> 2.4Mb（gzipしてない状態）
- 765kb -> 681kb（gzip）

参考

- rstacruz/webpack-tricks: Tips and tricks in using Webpack
- th0r/webpack-bundle-analyzer: Webpack plugin and CLI utility that represents bundle content as convenient interactive zoomable treemap

- Webpack & React Performance in 16+ Steps
- JavaScriptライブラリ/プロジェクトのファイルサイズの問題点を見つける方法 - Qiita
- Webpackバンドルを占めるサイズがでかい奴を探す方法 - Qiita

"module"フィールド対応

パッケージはBundleを配布しないで重複した依存ライブラリは削減できました。

Tree Shaking

webpack、rollup.js、ParcelといったBundlerではTree shakingと呼ばれる不要なコードを削除する仕組みを実装しています。

- Tree Shaking
- Reduce JavaScript Payloads with Tree Shaking | Web Fundamentals | Google Developers

Tree Shakingはモジュール間の構造を静的に解析して（副作用がない）不要なコードを削除する手法です。

Tree Shakingを行うには、モジュール間の依存関係を静的に解析できるようにしなければなりません。 CommonJSで広く使われているrequire()とmodule.exportsでのモジュール定義はただの関数とオブジェクトで定義であるため動的です。

動的とは実行してみなければそれがインポート/エクスポートできるかわからないものと言い換えられます。つまり、CommonJSでは次のような動的なインポートも問題ありません。

```
let myModule;
try{
    myModule = require("./myModule");
}catch(error){
    myModule = "default value"
}
```

このように動的なインポート/エクスポートは実行してみなければわからないので、静的な解析を行いモジュール間の依存関係を分析するのが難しいです。

一方で、ES2015で構文として導入された `import` と `export` 構文は実行する前に依存関係が分析できます。 なぜなら、 `import` や `export` はトップレベルのスコープ（モジュールスコープ）の直下に記述しないとエラーとなる構文であり、静的に依存関係が解析できるように設計されています。

つまり、次のような記述は構文エラーとなります。

```
let myModule;
try {
    import myModule from "./myModule";
}catch(error){
    myModule = "default value"
}
```

そのため、ほとんどのbundlerがTree Shakingの対象にできるのは `import` 、 `export` 構文を使っているコードのみです。

簡単にまとめると、Tree Shakingを行うには `import` と `export` の構文のままコードを含めたライブラリとして公開する必要があります。 Tree Shakingによって使われてないコードを削除するなどで、ファイルサイズやJavaScriptのパース速度の改善できます。

- JavaScript Start-up Optimization | Web Fundamentals | Google Developers

Tree Shakingの詳細は次の記事でも解説されています。

- 2018 年の tree shaking | 株式会社カブク

Tree Shakingに対応する

Tree Shakingに対応するには、そのライブラリが `import` 、 `export` 構文で出力しているコードを持っていることをbundlerに伝える必要があります。 この通知方法は標準化されたものではないですが、多くのbundlerは `package.json` の `"module"` フィールドを参照します。

通常のnpmで配布しているライブラリは `"main"` フィールドにかかれているパスを読み込みます。 この `"main"` フィールドにかかれているファイルはCommonJSの `require` で読めるファイルとなります。

`"module"` フィールドに書くファイルパスは名前のとおりECMAScript モジュール形式のコードで、このコードは `import` / `export` のままのコードを配置します。webpackなどのbundlerは `"module"` フィールドが存在する場合はそちらを優先して読み込みます。 （webpackではこの読み込む優先度をresolve.mainFieldsで変更できます）

そのため、管理しているライブラリそれぞれ `"module"` フィールドに対応する必要があります。(`"module"` フィールドに対応してないライブラリは今までとおり扱えるので、無理にすべてを対応する必要はありません）

次のような内容でIssueを作成し、対応するべきモジュールを一個ずつ対応していきました。 （ `"module"` フィールドは混在できるので、すべてを一気にやる必要はありません）

最終的には、この `"module"` フィールドの対応によってアプリのbundleのサイズは⬇65kb（gzip）ほど削減できました。 内訳としては外部ファイル（chunk）にしたSVGが20kb（gzip）、Tree ShakingとModuleConcatenationPluginによる削減が45kb（gzip）ほどでした。

↓ここから先はすべて当時書いたIssueの内容です↓

用語

- アプリ: アプリケーション – ライブラリを使う側（distを吐く）
- ライブラリ: `lib/` を持っていて使われる側のパッケージ

- "module"フィールド: package.jsonの"module"フィールドのこと
  - "main"とはことなり、ES module形式のモジュールへのパスを指定する
  - Rollupやwebpackが対応していてTreeShakingやDynamic Importを行うには必要
  - pkg.module · rollup/rollup Wiki

目的

- SVGロゴのReactコンポーネントを遅延ロードする（メインアプリのbundleからSVGを外す）
- webpackのbundleの最適化を有効化しファイルサイズやランタイムコストを小さくする
  - CommonJSは動的、ES Moduleは静的に解決される
  - この違いによりwebpackのTree ShakingやCode Splittingは、ライブラリ側でES Module形式の出力が必要となる（これは"module"フィールドに対応するということ）
  - Tree Shaking
  - Code Splitting
  - 2018 年の tree shaking | 株式会社カブク

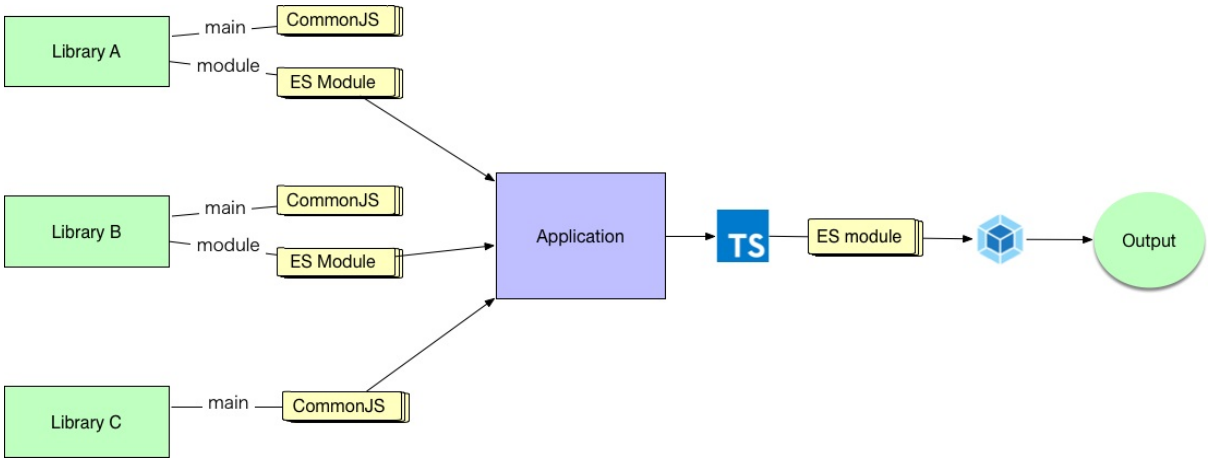
webpack 4からデフォルトのModuleConcatenationPluginの処理でランタイムのコストとファイルサイズが削減できる。 この最適化処理はES modulesのコードベースでないとできない。

- webpackのbundle時に、モジュールごとに安全のための無駄な即時実行関数によるスコープ作成が作成される
- モジュールの数が増えるほどコストが線形的に増えてしまう問題が発生してしまう
- The cost of small modules | Read the Tea Leaves この問題がModuleConcatenationPlugin相当の処理で軽減できる

必要なこと

各ライブラリ（使われる側）とアプリ側（使う側）でそれぞれES module形式のままwebpackにモジュールを渡せるようにする

最終的にこんな感じの構成になります。



- Library AやBは"module"フィールドに対応した例
- Library Cは"main"フィールドだけのCommonJsなライブラリが混ざっても問題ない
- ApplicationはTypeScriptではES module形式で出力して、最終的にはwebpack側でブラウザ向けに出力するのでES moduleがブラウザが読み込まれるわけではない

ライブラリ側

次の2つに対応する必要があります

- "module" フィールドに対応
  - `package.json` に "module"フィールドの追加
  - ES module形式でのライブラリ公開
- "sideEffects" フィールドに対応
  - `"sideEffects": false` : ライブラリはpolyfillなどglobalに影響を与えるモジュールを含んでいない
  - `"sideEffects": true` : ライブラリはpolyfillなどglobalに影響を与えるモジュールを含んでいる
  - `sideEffects:false` の場合は不要なモジュールが消せる

"module" フィールドへの対応方法

TypeScriptを利用してい場合は次のようなtsconfig.jsonの"module"向けの設定を追加する

- lib向け: `tsconfig.json`
- module向け: `tsconfig.json`

`tsconfig.module.json` :

```
{
  "extends": "./tsconfig.json",
  "compilerOptions": {
    "target": "es5",    // <= module以外は通常通りES5相当の出力になる
    "module": "esnext", // <= "module"によってimport/exportのまま出力できる
    "moduleResolution": "node",
    "outDir": "./module" // <= 通常のlibとは異なる場所に出力する
  },
  "include": [
    "src/**/*"
  ],
  "exclude": [
    "node_modules"
  ]
}
```

ビルド時はそれぞれのtsconfig.jsonで `tsc` を使ったビルドすればよい。

```
tsc -p ./tsconfig.json && tsc -p ./tsconfig.module.json
```

Note: tsconfig.jsonは複数の形式をサポートしてないため、２つの設定を用意して2種類のビルドを行う手法を取る。

- Suggestion: tsconfig.json support multiple configurations. · Issue #11172 · Microsoft/TypeScript

最後に `package.json` に"module"フィールドを追加する。 これでライブラリ側の"module"フィールド対応はひとまずできる

```
{
  "main": "lib/index.js", // <= ES5 + CommonJS
  "module": "module/index.js", // <= ES5 + ES module
  // ....
}
```

最終的には次のようなディレクトリ構造になる。

```
├── dist/
|      bundleしたファイル
├── lib/
|      src/ を JavaScript に変換したファイルと型定義ファイル
├── module/
|      src/ を JavaScript に変換したES module版と型定義ファイル
```

```
├── src/
|       TypeScriptのソースコード
```

"sideEffects" フィールドに対応

"sideEffects"フィールドに対応するとwebpackがさらに最適化できる。 詳細はTree Shakingにかかれている。 副作用（globalの挙動を書き換える）ものが明示されていれば、全く使ってないファイルを完全に消すことができる。デフォルトでは副作用があるかはわからないため、まったくつかってなくてもファイルを消すことができない。（importしただけで挙動を変更するモジュールがあるとだめになるため安全に倒す）

2018 年の tree shaking | 株式会社カブクに実際の例が紹介されている。

- `"sideEffects": false`：ライブラリはpolyfillなどglobalに影響を与えるモジュールを含んでいない
- `"sideEffects": true`：ライブラリはpolyfillなどglobalに影響を与えるモジュールを含んでいる

に従った値を指定する。 特定のファイルだけ副作用があるという設定もできる。

```
{
  "name": "your-project",
  "sideEffects": [
    "./src/some-side-effectful-file.js",
    "*.css"
  ]
}
```

副作用なしのモジュールは消える場合があるので、`css-loader`などでCSSがタグに対してスタイルを当てる（副作用がある）場合はCSSを副作用ありにしないといけない。

アプリ側

`"module": "esnext"`への対応

利用する側もwebpackにはES Moduleのまま渡すようにする。

TypeScriptならばアプリ側の`tsconfig.json`も`"module": "esnext"`に対応する。 これでアプリ側のコードがES moduleとなりwebpackなどが最適化できる。

`tsconfig.json`：

```
{
  "compilerOptions": {
    // アプリ側なので他にもいろいろ設定があるはず
    // ... 省略 ...
    "target": "es5",    // <= module以外は通常通りES5相当の出力になる
    "module": "esnext", // <= "module"によってimport/exportのまま出力できる
    "moduleResolution": "node",
    "outDir": "./lib",
    // ... 省略 ...
  "include": [
    "src/**/*"
  ],
  "exclude": [
    "node_modules"
  ]
}
```

Code Splittingへの対応

ライブラリ側でDynamic Importをしていて、かつwebpack向けにマジックコメントを利用しておけば、 アプリ側でCode Splittingの対応するだけで動的ロードに対応できる。

```
// lodashという`[name]`のchunkになる
import(/* webpackChunkName: "lodash" */ 'lodash').then(lodash => { /* * /})
```

- Code Splitting

アプリ側ではwebpack.config.jsで `chunkFilename` の対応を行えばよい。

```
  const path = require('path');

  module.exports = {
    mode: 'development',
    entry: {
       index: './src/index.js'
    },
    output: {
      filename: '[name].bundle.js',
      // ライブラリ側のDynamic Importもchunkに吐き出せる
      chunkFilename: '[name].chunk.js',
      path: path.resolve(__dirname, 'dist')
    },
  };
```

`chunkFilename` が指定されていれば、次のように書いた"lodash"モジュールは `lodash.chunk.js` というファイルに分割して出力され、実行時に動的ロードされる。

```
// lodashという`[name].chunk.js`のchunkファイルが作成される
import(/* webpackChunkName: "lodash" */ 'lodash').then(lodash => { /* * /})
```

chunkのファイル名

ライブラリ側では読み込むchunkを `import(/* webpackChunkName: "ファイル名" */ "module")` というマジックコメントで指定する必要がある。

- Code Splitting
- Module Methodsを参照

この `ファイル名` は必ずしもユニークである必要はないが、chunkとして分けたときにわかりやすい名前（URL）になるのを想定して決めること。

実際にchunkファイルを生成する側の設定で、出力するファイル名規則を設定できる。 次のようにchunkファイルのハッシュ値をファイル名に入れることができるため、ファイル名自体は被った際にも、別ファイルとして出力できる。

```
chunkFilename: 'scripts/chunk.[name].[chunkhash:10].js'
```

読み込むファイル名を `webpackChunkName` に指定すれば大きな問題はなさそう。

publicPathの設定

webpackでビルドする際に `publicPath` にあたる情報が適切に設定されていない、Dynamic Importしたファイルが意図しないURLを参照してしまう問題。 (chunkに分離はできたけど、 `import("./chunk.js")` がNot foundとなってしまうようなケース)

次のようにbundle.jsから `import("./chunk.js")` を読み込む場合に、 `publicPath` をベースとして相対パス(`./chunk.js`)を読み込む。 そのため、次のようにウェブページのURLとJavaScriptファイルを置くCDNのドメインが異なる場合などは、 `bundle.js` から `./chunk.js` を読み込むことができない。

ウェブサイトとCDNでドメインが異なる場合の例:

- `https://example.com/`
  - `<script src="https://cdn.example.com/bundle.js"></script>`
- `https://cdn.example.com/bundle.js`
  - `import("./chunk.js")` -> `https://example.com/chunk.js` は 404になる
- `https://cdn.example.com/chunk.js`

この場合、 `publicPath` に当たる情報には `https://cdn.example.com/` が指定されていないと、 `./chunk.js` を `https://cdn.example.com/chunk.js` として解釈できずにDynamic Importは失敗してしまう。

Public Pathにかかれているように次の2つのどちらかを使って、 `publicPath` の値を指定する。

- コンパイル時に、 `publicPath` に CNDのAssertまでのパスを指定する
- 実行時に、 `__webpack_public_path__` へCNDのAssertまでのパスを指定する

すでにアプリ側で実行時にAssetまでのパスを持っているなら、必ず通る場所に `__webpack_public_path__` の代入を行うが手早い。 (`publicPath` はいろいろな環境に分岐するには、webpack.configで分岐を書く必要がでてくるため)

```
__webpack_public_path__ = assertBaseURL
```

先程の例ならば次のように指定されていれば、Dynamic Importでchunkを読み込むことができる。

```
__webpack_public_path__ = "https://cdn.example.com/"
```

FAQ

よくある質問集

Dynamic Importをchunkにしたくない場合

"module"対応したライブラリを利用していて、かつそれをラップしたライブラリをbundleとして配布したい場合について（ライブラリでbundle版を配布したい場合）

tl;dr: `LimitChunkCountPlugin` プラグインを使うとwebpackはchunkを勝手にわけない

`import()` を使っているライブラリがあるとwebpackは自動的にそれをchunkに分けて出力する。 （moduleで対応しているとDynamic Importが認識されるのでchunkに分ける）

bundleしたライブラリ（Dynamic Importなし）として配布したい場合は、 `LimitChunkCountPlugin` を使うことでchunkへの分離を抑制できる。

```
new webpack.optimize.LimitChunkCountPlugin({
  maxChunks: 1
})
```

- How to disable Code Splitting in webpack – Glenn Reyes – Medium

"module"対応したライブラリを使いたくない場合

"module"フィールド（ES Modules）ではなく今までと同じ"main"フィールド（CommonJS）としてライブラリを使いたい場合について。

tl;dr: `mainFields: ["main"]` を指定する

webpackのresolve.mainFieldsを設定することで、ライブラリのどのフィールドを優先的に使うか（ES ModulesかCommonJSか）をアプリ側で指定できます。

デフォルトでは、次のように `module`（ES modules）> `main`（CommonJS）の優先度に指定されています。

```
resolve: {
    mainFields: ['browser', 'module', 'main'],
    // 他の設定
},
```

次のように `main`（CommonJS）のみに指定をすれば、ライブラリが"module"フィールドに対応していた場合でも、"main"フィールドを利用するように設定できます。

```
resolve: {
    // module フィールドは使わない
    mainFields: ['main'],
    // 他の設定
},
```

ファイルサイズを減らす

ページロードを早くすることにおいてファイルサイズを小さくすることは重要です。 特にJavaScriptのようなパースやコンパイルといった処理を必要するスクリプトのファイルサイズを減らすことは、 そのまま処理コストにも影響します。

次の記事でもあるように同じサイズのJavaScriptと画像は同じコストではありません。

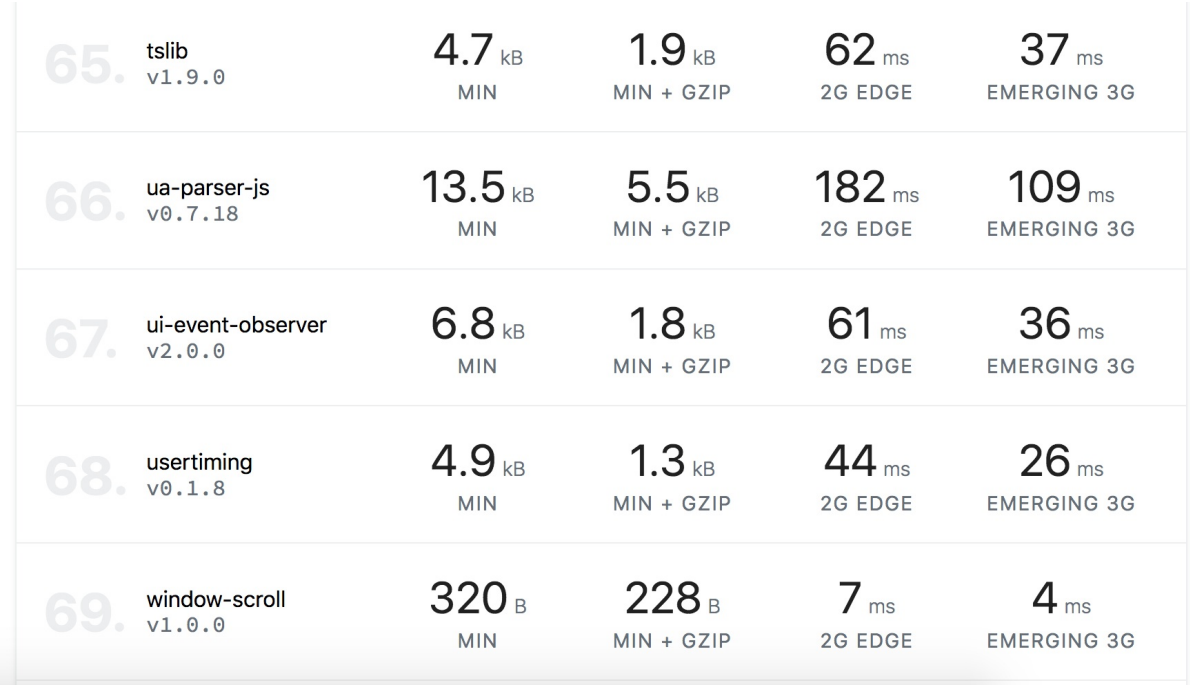
- The Cost Of JavaScript In 2018 – Addy Osmani – Medium

しかしながら、ファイルサイズを小さくするにはアプリケーションによってさまざまなパターンが考えられます。
ここではいくつかのアプローチについて見ていきます。

Bundleを分析する

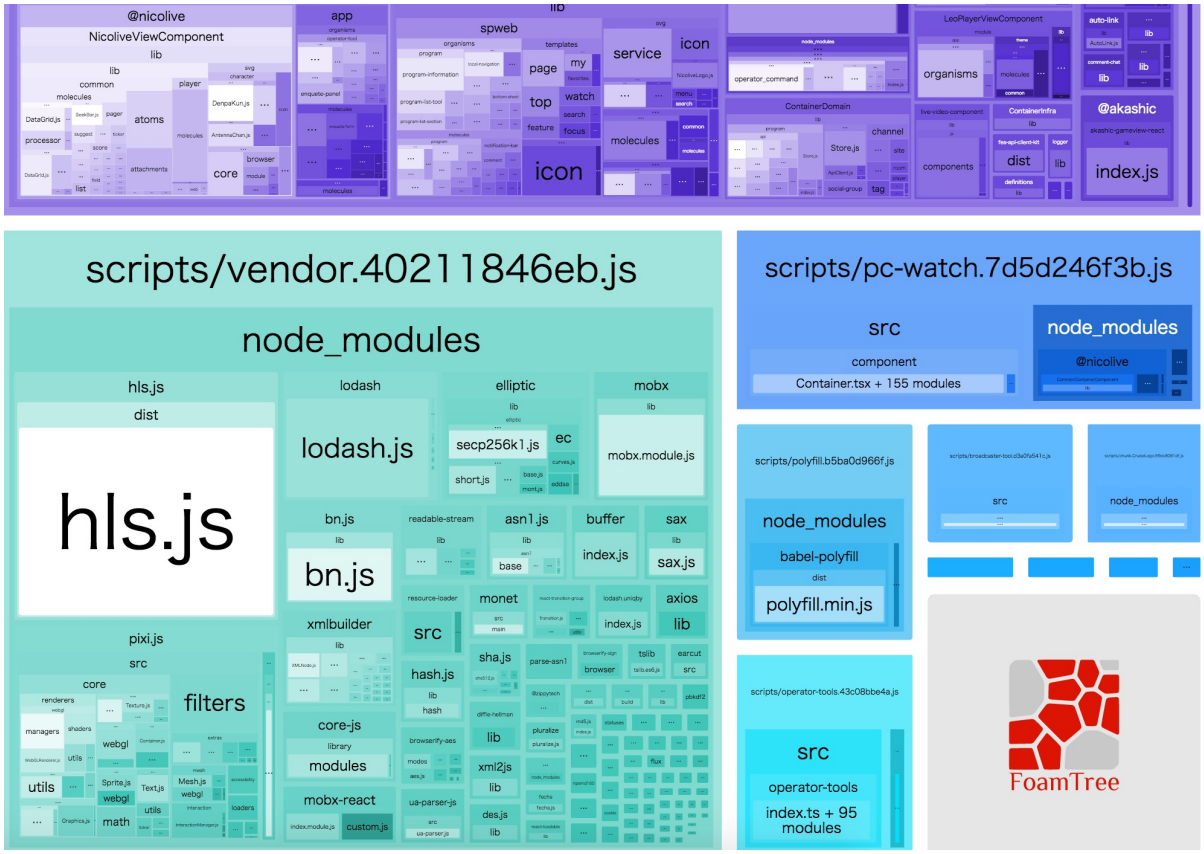
これは単純ですがWebpackなどでbundleしたJavaScriptファイルを分析するというアプローチです。 ここで見つける問題としては意図せずに入ってしまっているライブラリ、想像より大きなライブラリなど含まれていないかです。

特定のライブラリのサイズを見るにはpackage-sizeやbundlephobiaなどを使い、そのライブラリの依存を含めてサイズを見ます。 そのライブラリ自体のコード量ではなく、必ずbundleしてminifyしたgzipのサイズなどで比較します。 なぜなら、ライブラリ自体のサイズにはコメントなど圧縮すると大きくサイズが変わるものや、ライブラリのコードは少なくても依存してるライブラリのサイズが大きいという問題があるためです。



> bundlephobia.com/scanで `pacakge.json` に書かれたライブラリのサイズを一覧

すでにアプリケーションのコードをWebpackでbundleしている場合はwebpack-bundle-analyzerを使うとサイズがわかりやすく可視化できます。



webpack-bundle-analyzerでモジュールのサイズを可視化した例

webpack-bundle-analyzerなどは、環境変数などで普段のビルド + bundle-analyzerの結果を出力できるようにwebpackの設定ファイルを作成しておくとよいです。

例) `BUILD_STATS=1 webpack` のように環境変数でwebpack-bundle-analyzerを有効化する

```
const path = require('path');
const webpack = require('webpack');
const BundleAnalyzerPlugin = require('webpack-bundle-analyzer').BundleAnalyzerPlugin;
const BUILD_STATS = !!process.env.BUILD_STATS;
module.exports = {
    // ...省略...
    plugins: [].concat(
      BUILD_STATS
       ? [
        new BundleAnalyzerPlugin({
          analyzerMode: 'static',
          reportFilename: path.join(__dirname, './build/stats/app.html'),
          defaultSizes: 'gzip',
          openAnalyzer: false,
          generateStatsFile: true,
          statsFilename: path.join(__dirname, './build/stats/app.json'),
          statsOptions: null,
          logLevel: 'info'
        })
      ]
      : []
      )
    )
    .concat([
      new webpack.ContextReplacementPlugin(/moment[\\/]locale$/, /^\.\/(ja)$/),
      new LodashModuleReplacementPlugin({ shorthands: true })
    ])

}
```

webpack-bundle-analyzerを使った分析とファイルサイズの削減については、次の記事を参照するとよいでしょう。コードをminifyするといった基本的なことからmomentのようなよくある大きなライブラリの扱い方に、より高度なファイルサイズの削減方法などについて書かれています。

- Put Your Webpack Bundle On A Diet - Part 1
- Put Your Webpack Bundle On A Diet - Part 2
- Put Your Webpack Bundle On A Diet - Part 3
- Put Your Webpack Bundle On A Diet - Part 4

より小さなライブラリに変更してサイズをへらすことも重要ですが、まずは使ってないライブラリや不用意に入ってしまっているものを取り除くことから始めるのがよいです。

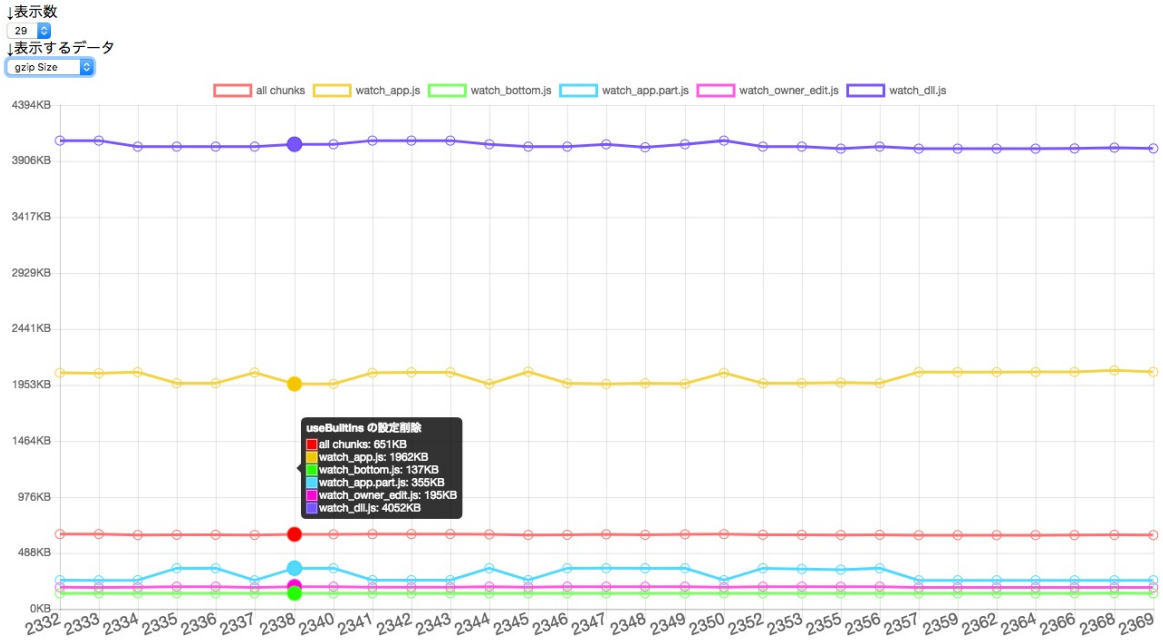
例) よくある問題

- 意図せずに含まれているライブラリを取り除く
- 異なるバージョンが複数含まれているのを1つに統一する
  - duplicate-package-checker-webpack-plugin
- momentの不要なロケールを取り除く
  - moment-locales-webpack-plugin
- lodashで未使用なメソッドを取り除く
  - lodash-webpack-plugin

アプリケーションのサイズ（特にJavaScript）はライブラリによって大きく増減します。 そのため、突発的なファイルサイズの増加を防ぐためにもbundlesizeのようなファイルサイズチェックをPRごとに行ったり、size-pluginで現在のファイルサイズをビルド時に可視化するといった工夫も必要になります。

ファイルサイズの変化をどのように検知するかはいろいろなトレードオフがあるので、チームにあったものを選択するのがいいと思います。

たとえば、あるチームではPRごとにwebpack-bundle-analyzerの結果をグラフとして見られるようにして、ファイルサイズの変化を見ていました。



もちろん合成モニタリングサービスを使った計測も併用します。 しかし、外部からの監視は実際にデプロイするまでわからないため、このようなPRやコミットなど複数のレイヤーでチェックを併用します。 併用して自動チェックできるレイヤーを増やすことで問題に早く気づくことができるようになります。

初期表示に必要ないものを遅延ロードする

もっと単純で効果があるアプローチは、必要がないものは読み込まないことです。 注意しないと初期表示に不要なJavaScriptやCSSを含んでしまう場面も多いと思います。

これは特にサードパーティスクリプトの読み込みなどが該当しやすいです。 JavaScriptなら `async` 属性を付けて遅延ロードさせ、loadCSSなどを使って遅延ロードさせるという手法が利用できます。

```
- <script src="https://example.com/script.js" />,
+ <script src="https://example.com/script.js" async defer />,
```

- レンダリングを妨げる JavaScript を削除する | PageSpeed Insights | Google Developers
- クリティカル リクエスト チェーン | Tools for Web Developers | Google Developers
- Modern Asynchronous CSS Loading | Filament Group, Inc., Boston, MA

特定のコンポーネントに紐付いて必要となるスクリプトなども、そのコンポーネントが表示されるまで読み込む必要がないはずです。 これは、広告のスクリプトやSNSボタンなど特定のコンポーネントに紐付くものなどが該当します。

これらのコンポーネントに紐づくスクリプトなどは、動的にロードしてそれが読み終わったタイミングでコンポーネントを更新するといった作りにすることで遅延ロードできます。

次のコードは、動的にスクリプトをロードするUtilと、スクリプトをロードするReactコンポーネントの例です。

`ScriptLoaderUtil.ts`：

```
/**
 * <script>タグでjsファイルを動的にロードするUtil
 */
export class ScriptLoaderUtil {
  /**
   * すでに読み込みを開始したURLを保持しておくマップオブジェクト
   */
  private static srcMap: { [src: string]: Promise<void> } = {};

  /**
   * デフォルトのタイムアウト(ミリ秒)
   */
  public static DEFAULT_TIMEOUT = 5 * 1000;

  /**
   * scriptタグを利用して外部ソースの読み込みを行う
   * @param src 読み込むjsのURL
   * @param timeout タイムアウト(ミリ秒)
   */
  public static load({ src, timeout }: { src: string; timeout?: number }): Promise<void> {
    return new Promise((resolve, reject) => {
      // すでに読み込みを開始している場合は2重に読み込まない
      if (this.srcMap[src]) {
        return this.srcMap[src].then(resolve);
      }
      // 新規 src の場合は script タグを利用して読み込みを開始する
      this.srcMap[src] = new Promise((resolve, reject) => {
        let isHandled = false;
        const script = document.createElement('script');
        script.src = src;
        script.type = 'text/javascript';
```

```
        script.charset = 'utf-8';
        script.async = true;
        /* tslint:disable */
        const onLoad = () => {
          isHandled = true;
          resolve();
          script.removeEventListener('load', onLoad);
          script.removeEventListener('error', onError);
        };
        const onError = () => {
          isHandled = true;
          reject(new URIError(`The script(${src}) is not accessible.`));
          script.removeEventListener('load', onLoad);
          script.removeEventListener('error', onError);
          delete this.srcMap[src];
        };
        /* tslint:enable */
        script.addEventListener('load', onLoad);
        script.addEventListener('error', onError);
        document.body.appendChild(script);
        // タイムアウト処理
        setTimeout(() => {
          if (isHandled) {
            return;
          }
          script.removeEventListener('load', onLoad);
          script.removeEventListener('error', onError);
          delete this.srcMap[src];
          reject(new Error(`The script(${src}) load is timeout`));
        }, timeout || this.DEFAULT_TIMEOUT);
      });
      return this.srcMap[src].then(resolve, reject);
    });
  }
}
```

`ScriptLoader.tsx`:

```
import * as React from 'react';
import classNames from 'classnames';
import { ScriptLoaderUtil } from './ScriptLoaderUtil';

export type ScriptLoaderProps = {
  src: string;
  className?: string;
  onLoad?: () => void;
  onError?: (error: Error) => void;
};

/**
 * <script> で JS ファイルを読み込むコンポーネント
 * <ScriptLoader src="https://example.com/script.js" onLoad={() => {}} />
 */
export default class ScriptLoader extends React.PureComponent<ScriptLoaderProps> {
  public componentDidMount() {
    ScriptLoaderUtil.load({
      src: this.props.src
    })
      .then(() => {
        this.props.onLoad && this.props.onLoad();
      })
      .catch(error => {
        if (this.props.onError) {
          this.props.onError(error);
        } else {
          console.error(error);
        }
```

```
      });
  }

  public render() {
    return <div className={classNames('ScriptLoader', this.props.className)} />;
  }
}
```

分離と自動チェック

webpackなどではCode Splittingで初期表示に必要ないものをbundleから別のchunkに分割できます。

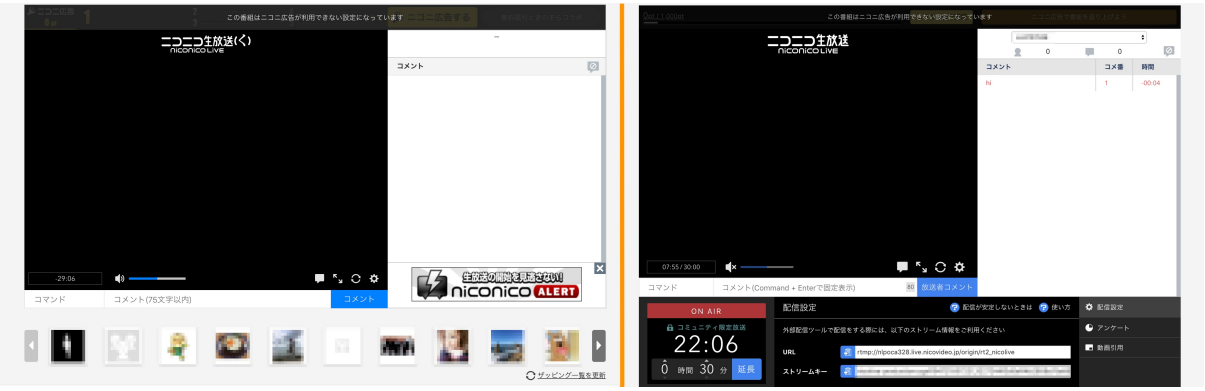
- Code Splitting
- Code-Splitting – React

これによって、初期表示に必要なbundleのファイルサイズを削減できます。

どのように分割するかはアプリケーションに依存しますが、もっとよくあるケースはルーティングによってページ（URL）として別れているコンポーネントを分割することです。 また、同じページ内でも一般ユーザーと管理ユーザーで表示されるものが異なるということがあります。 管理ユーザーでは"一般ユーザーの表示" + "管理ツール"といった構造になっていることも多いです。

このような場合にメインのbundleに一般ユーザーに不要な"管理ツール"のコンポーネントを含めるのは適切ではありません。 そのため、"管理ツール"だけをchunkとして分けたり、Dynamic Importsで動的にロードするといったことができます。

具体的な例として、ニコニコ生放送では視聴者がみる視聴ページと配信者が見る配信ページはほとんど同じJavaScriptで動作しています。 しかし、配信者が見るページには、視聴者見るプレイヤーなどに加えて配信者向けのツール（コンポーネント）が追加されています。



> 左は視聴者が見るページ、右は配信者が見るページ

この"配信者向けのツール"は視聴者に不要であるためメインのbundleから外す必要ことでファイルサイズが削減できます。

実際にこの"配信者向けのツール"を、メインのbundleから配信者向けのツールだけのchunk（ファイル）へと分離しました。 これによってメインのbundleが"配信者向けのツール"の分である30kb（gzip）程度削減できました。

この変更では、次のようにイメージで利用できるように分離されました。

```
<script src="main.bundle.js">
if (配信者ページなら) {
    <script src="配信者向けツールのbundle.js">
}
```

bundleの再結合の防止

Bundleを分離することはうまくできても、その状態を維持する必要があります。 たとえば、先ほどの"配信者向けのツール"の分離では、ディレクトリも次のように分離していました。 そしてそれぞれに対応するbundleとchunkを出力するようになっています。

```
src/
├── broadcaster-tool(配信者向けのツールのコード) ---> broadcaster-tool.jsとして出力
└── view(視聴者向けのコード)  ---> pc-watch.jsとして出力
```

これがうまく分離されるには `view(視聴者向けのコード)` と `broadcaster-tool(配信者向けのツールのコード)` の間に依存関係を作らない必要があります。 なぜなら、 `view` が `broadcaster-tool` に依存関係があるとwebpackなどはchunkとして分離できなくなるためです。 たとえば、 `view(視聴者向けのコード)` から `broadcaster-tool(配信者向けのツールのコード)` に依存がある `pc-watch.js` に `broadcaster-tool.js` の内容が含まれてしまいます。

これは、JetBrains系のIDEやVSCodeなどJavaScriptのモジュールへのパス補完が簡単できるといつの間にか起きてしまうことがあります。 コンポーネント名を入力すると、自動的にそのコンポーネントを `import` してしまって想定外の依存関係が発生すると言ったことがおきます。

このような意図しない方向の依存を防止するにはdependency-cruiserのようなチェックツールを使うのが簡単です。 `dependency-cruiser` はルールを書いて、モジュールの依存関係のチェックを行えます。

- sverweij/dependency-cruiser: Validate and visualize dependencies. With your rules. JavaScript, TypeScript, CoffeeScript. ES6, CommonJS, AMD.

具体的には、次のようなルールを書くと `src/broadcaster-tool` -> `src/view` への参照を自動的にチェックして、意図しない参照を防止しています。

```
{
    "forbidden": [
        {
            "name": "broadcaster-tool-does-not-import-view-index",
```

```
            "comment": "broadcaster-toolはview/index.tsを参照するとcode splitできなくなる。分割したbroadcaster-toolにview/index.tsから参照するすべてのファイルが含まれてしまうのを避けるため、直接view以下のファイルを参照してください",
            "severity": "error",
            "from": {
                "path": "^src/broadcaster-tool"
            },
            "to": {
                "path": "^src/view"
            }
        }
    ],
    "options": {
        "doNotFollow": "node_modules"
    }
}
```

類似サービスと比較する

ファイルサイズは大きくても動かなくなるわけではないのため、どこに原因があるかがわかりにくいという問題があります。 そのときの判断材料として、類似するサービスと比較して見る方法があります。 (SpeedCurveなどではCompetitorのURLと一緒に計測して、結果を比較といった方法もできます。)

たとえば、ニコニコ生放送ならばニコニコ動画、Youtube、FRESHLIVEなど類似する機能をもつサービスはいろいろあります。 それらのサイトと比較のファイルサイズを比較して、問題を見つけるというのも1つの手段です。

JavaScriptのサイズはもつ機能や作りによって大きな差はでます。 一方、CSSは画面を構成する要素に依存しまた想定する画面サイズにはそこまで差はでないため、大きな差が出にくいとも考えられます。

ニコニコ生放送の視聴ページのCSSを他のサイト比較してみると、なぜか3倍程度大きいという問題ことがわかりました。 CSSが大きくなっている原因を次のツールなどで調べてみると、いくつかの問題があるということがわかりました。

- TestMyCSS | Optimize and Check CSS Performance
- CSS Stats

具体的に見つかった問題は次のとおりです。

- Base64 Lengthがでかい（Base 64のサイズが大きい）
  - 前者はCSSに画像やfontがBase64でそのまま埋め込まれていた
- Empty Rulesの数が多い（空のセレクタにコメントだけが残っている問題があった）

これらの問題に対してそれぞれ次のような解決方法を取りました。

- 画像やfontなどのリソースをBase 64ではなくURLとして指定するようにした
  - fontは実際に使用されるまでダウンロードされなくなった
- cssnanoのdiscardCommentsルールを適用してコメントを消して、minifyで空セレクタがきえるようにした

CSSはSassやPostCSSなどのさまざまなツールを経由して出力されることが多いため、不用意にサイズが増えてしまっていることがあります。 そのため、Performance budgetsを設定するなど不自然な増え方をした場合に気づく仕組みが必要になるでしょう。