My Web ちぇんじろぐ

2022年 part2

ver. 1

2022年7月から2022年12月までに
登場したライブラリ・フレームワーク
サービスを好き勝手紹介する本

はじめに

本書は2022年7月から2022年12月末までに登場したWeb関連の出来事を著者の趣味でまとめた本になります。何かしらの新しい出会いがあることを祈ってます。

本書で扱うこと

本書では以下のカテゴリーに分けて内容紹介してます。また、各内容は独立しているため、つまみ食いしながら読んでいただければ幸いです。

- WebAssembly編
- サービス編
- フレームワーク編
- ライブラリ編
- ツール編
- 番外編

問い合わせ先

誤植や間違い、その他感想などはTwitterやメールなどで教えていただけると助かります。また、本書ではわかりやすさ優先のため、エンドポイントやリソース名をマスクせずに表示してます。それらのリソースは削除済みなので、ご連絡不要です。

メールアドレス: pilefort2020@gmail.com

Twitter: @pilefort (https://twitter.com/pilefort)

サービス編

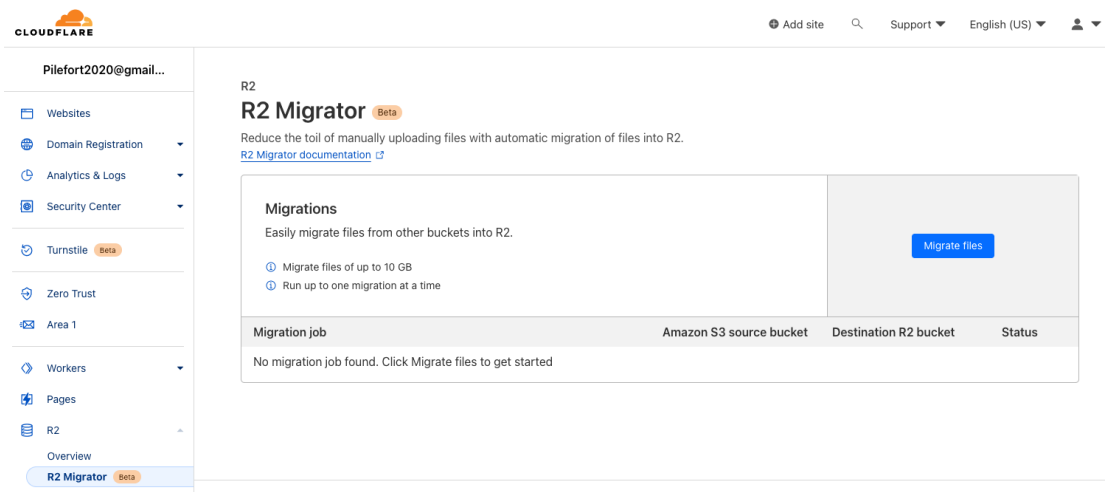
Cloudflare R2

検証日: 2023年5月6日

R2はCloudflareが提供するAmazon S3 API互換のストレージサービスです[1]。Amazon S3ではデータの読み取り書き込みだけでなく、データの転送自体も課金されますが、R2ではデータ転送料(egress charges) が無料になります。また、Cloudflare Workerで利用できるため、CDN Edgeからストレージにアクセスできます。

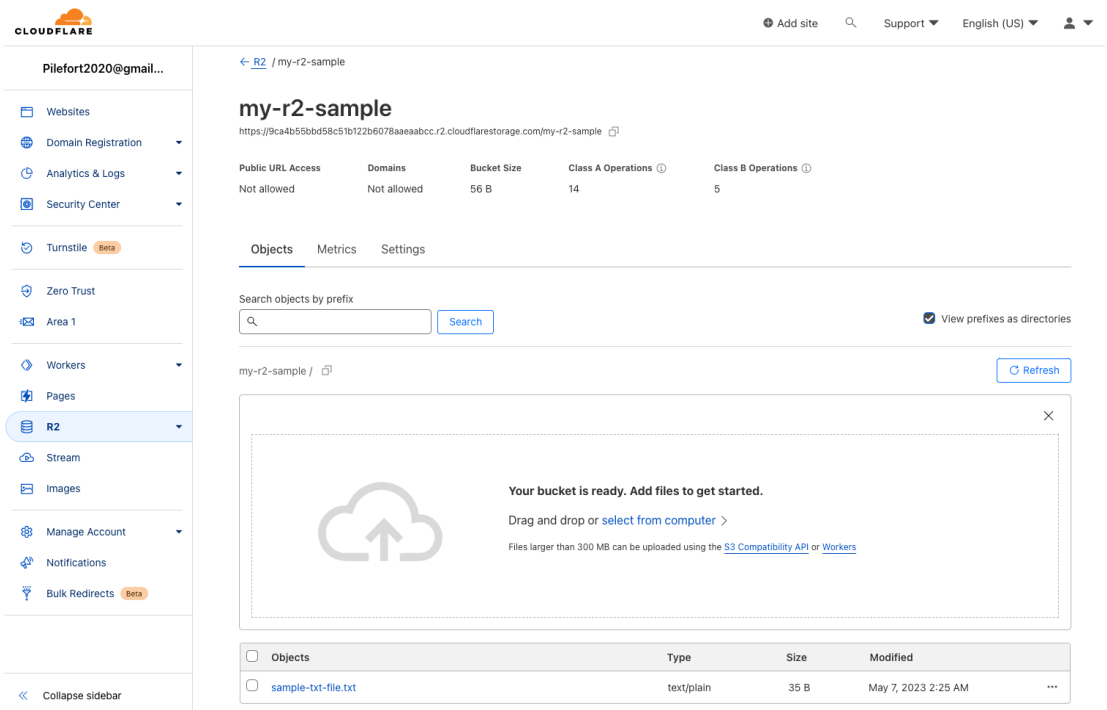
R2 Migrator

Amazon S3からCloudflare R2へ移行するためのサービスとして、R2 Migratorも提供されてます[2]。通常クラウド間のデータ移行をする際はrcloneなどのツールを使う必要がありますが、クラウド自体にデータ移行サービスを持っているのは大変使いやすいです。



Cloudflare R2 Migrator

ちなみに、ダッシュボードはこのようになっており、GUIでもファイルを保存できます。しかし、Amazon S3と異なり、ファイル数やデータ数を確認する機能はないので、データ移行後のチェックが少々大変そうです。



Cloudflare R2

Workerからのアクセス

以下のようなコードでWorker内からR2にアクセスできます。

wrangler.toml

```
name = "cloudflare-sample"
main = "src/index.ts"
compatibility_date = "2023-04-19"

[[r2_buckets]]
binding = 'R2'
bucket_name = 'my-r2-sample'
preview_bucket_name = 'my-r2-sample-development'
```

src/index.ts

```
export default {
    async fetch(request: Request, env: Env): Promise<Response> {
        const url = new URL(request.url)
        const objectName = url.pathname.slice(1)
```

```
    // list
    // ...
    // R2というのはwrangler.tomlで定義したbindingになります
    const listing = await env.R2.list()

    // GET
    // ...
    const object = await env.R2.get(objectName)

    // PUT/POST
    // ...
    const object = await env.R2.put(objectName, request.body)

    // DELETE
    // ...
    await env.R2.delete(url.pathname.slice(1))
    }
```

---

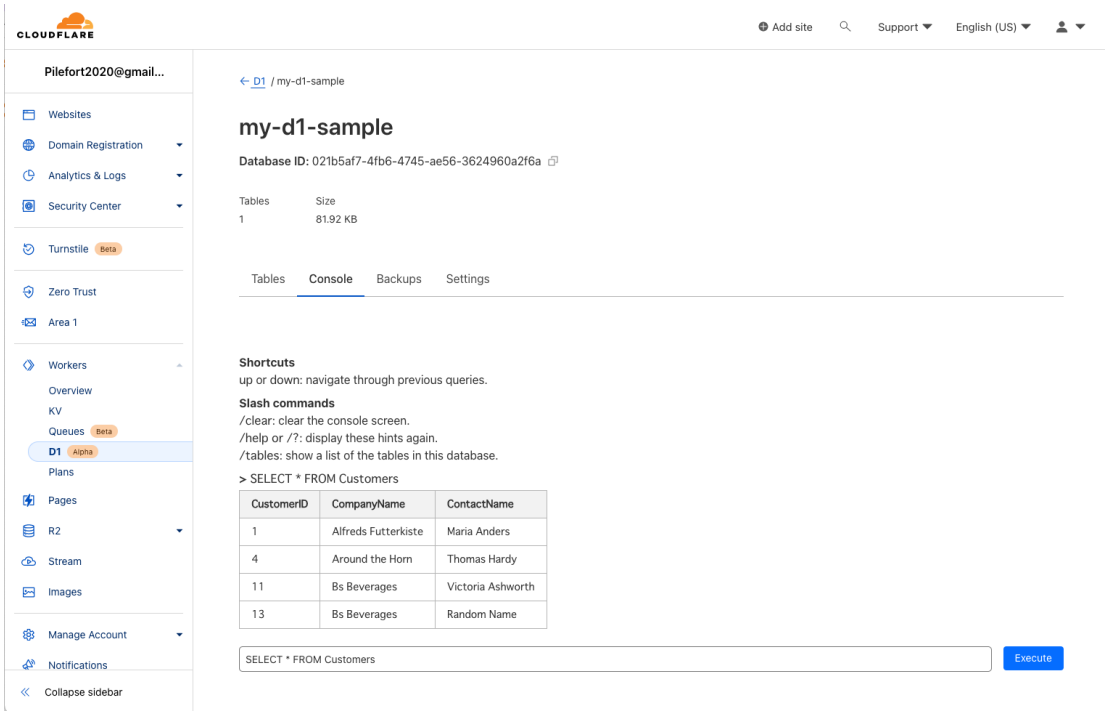
1. https://blog.cloudflare.com/r2-ga/↩
2. https://developers.cloudflare.com/r2/r2-migrator/↩

Cloudflare D1

検証日: 2023年5月6日

D1はSQLiteベースのCDN Edge (Cloudflare Worker) からアクセス可能なデータベースです[1]。ちなみに、データベースのバックアップはR2に保存されるらしいです[2]。SQLite -> オブジェクトストレージという流れがLitestreamに近い印象を受けますが、仕組みは異なるそうです[3]。

ダッシュボードではConsoleタブでクエリを発行できます。



Cloudflare D1のデモ (ダッシュボード)

workerからアクセス

workerからD1へのアクセスは以下のようにできます。クエリは生のSQLを書くか、コミュニティで準備しているORMを使います[4]。

wrangler.toml

```
name = "cloudflare-sample"
main = "src/index.ts"
compatibility_date = "2023-04-19"
```

```
[[ d1_databases ]]
binding = "DB"
database_name = "my-d1-sample"
database_id = "021b5af7-4fb6-4745-ae56-3624960a2f6a"
```

src/index.ts

```
export interface Env {
  DB: D1Database;
}

export default {
  async fetch(request: Request, env: Env) {
    const { pathname } = new URL(request.url);

    // DBはwrangler.tomlで定義したbinding
    const { results } = await env.DB.prepare(
      "SELECT * FROM Customers WHERE CompanyName = ?"
    )
    .bind("Bs Beverages")
    .all();

    return Response.json(results);
  }
};
```

---

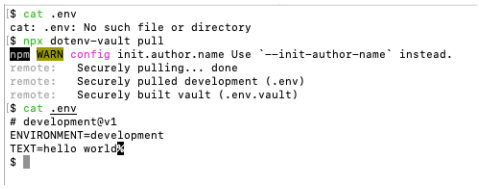
1. https://blog.cloudflare.com/introducing-d1/↩
2. https://developers.cloudflare.com/d1/↩
3. https://twitter.com/benbjohnson/status/1525149884541612033?lang=en↩
4. https://developers.cloudflare.com/d1/platform/community-projects/↩

dotenv vault

検証日: 2023年5月14日

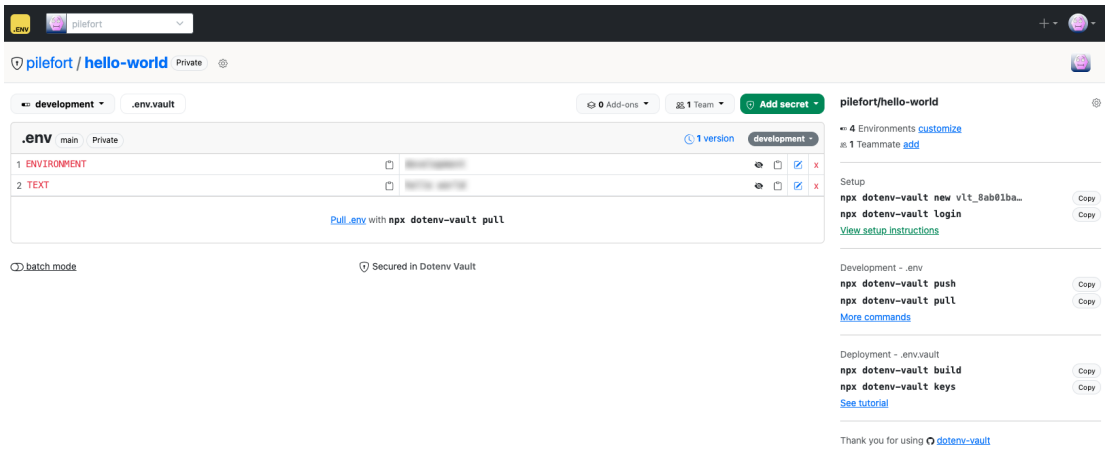
dotenv vaultは.envをクラウド上で同期、管理するためのサービスです[1]。リスクが高すぎて、なかなか使う気になれないですが、dotenv.orgが提供するサービスになります。

こちらの利用すると、gitのように.envを管理することができます。認証されたユーザーであれば、`npx dotenv-vault pull`でキー情報の取得ができ、`npx dotenv-vault push`でキー情報を更新できます。ちなみに、プロジェクト (フォルダ) とdotenv vaultの紐付けは`npx dotenv-vault new vlt_<プロジェクト固有のキー>`で実施します。



dotenv vaultのデモ

クラウド側ではこのように、プロジェクトごとに環境変数を保存できます。



dotenv vaultのクラウド側管理画面

dotenv vaultは1PasswordやLastPassなどの会社が携わらない限り、ちょっと怖くて使いづらいですが、便利なサービスではあるので、今後の展開に期待してます。

1. https://github.com/dotenv-org/dotenv-vault↩

フレームワーク編

tauri mobile

検証日: 2022年12月28日

tauriとは

tauriはデスクトップアプリケーションやモバイルアプリケーションを開発するためのフレームワークです。画面の見た目部分 (WebView) をJavaScriptで、バックグラウンド (メインプロセス側) をRustで記述できます。WebViewでは、React/Vue/Solid/Svelte/Next.jsなどが利用可能で、TailwindCSSなどのNPMライブラリも利用できます。メインプロセス側はRustで記述できるため、メモリ安全性やパフォーマンスを重視した開発ができます。

tauriとElectronの違い

デスクトップアプリケーションを開発できるフレームワークとしては、Electronも良く知られています。それぞれの違いは以下のようになります。

並列処理のやりやすさ

ElectronのメインプロセスはNode.jsです。Node.jsはシングルスレッドでの処理を前提としているため、大量のデータを並列処理しようとすると、複雑化します。対して、tauriはRustを使用しているため、メモリ安全に並列処理を書くための仕組みが用意されています。

実行環境の違い

Electronはアプリを出荷する際、Chromiumと一緒にパッケージ化します。対して、tauriは実行端末の標準ブラウザを自動認識し、そのブラウザ上で動作します。tauriは実行環境をパッケージ化しないため、アプリのサイズが小さくなります。

ちなみに、macOSではKWWebView, LinuxではWebkit2gtk, WindowsではWebView2上でtauriが動作します[1]。調べた限りだと現状 (2022年12月18日)、各標準ブラウザごとの差分を吸収またはテストする方法がないため、少し怖いですが、今後何かしらの手法が出てくることを期待してます。

セキュリティ面

tauriはスポンサーとして、1PasswordやCloudflareなどのセキュリティに強い企業が参加しています。また、tauri自体にもセキュリティチームが配備されており、セキュリティ上の問題の発見と対処に力を割いています。ElectronはChromium由来の脆弱性リスクがありますが、tauriはそのようなリスクがないだけでなく、セキュリティにも力を入れている点が安心できます。

tauri mobile (α版)

tauriの概略説明は以上にし、この度、α版ですが、tauriをモバイルアプリケーション上で動作できるようになりました[2]。React NativeのExpoのようなモバイルアプリをデプロイするためのワークフローはまだありませんが、今後の展開に期待しています。

ちなみに、WebView側とバックエンド側にそれぞれ以下のパッケージを追加するだけで、モバイルアプリを開発できます。

```
# WebView
$ yarn upgrade @tauri-apps/cli@next @tauri-apps/api@next
```

```
# バックエンド
cargo add tauri@2.0.0-alpha.0
cargo add tauri-build@2.0.0-alpha.0 --build
cargo install tauri-cli --version "^2.0.0-alpha"
```

---

1. https://tauri.app/v1/references/webview-versions/↩
2. https://tauri.app/blog/2022/12/09/tauri-mobile-alpha/↩

TailwindCSS 3.2

検証日: 2022年5月6日

TailwindCSSは3.0のときに劇的な変化を迎え、自由度の高いCSSフレームワークになりました。3.0以前は決められた単位でしかフォントサイズや色、パッディングを指定できませんでしたが、3.0からは任意のフォントサイズ、色、パディングを指定できるようになりました。

スタイルを当てる条件付け

TailwindCSS 3.2からは、ブラウザサポートやARIA属性、data属性、画面幅によってスタイルの当て外しができるようになりました。

例えば、ブラウザがそのCSSをサポートしてるかどうかを判定するなら、support-* variantを使います[1]。以下のコードでは`display:grid`がサポートされるかどうか判断し、サポートしてる場合は`display:grid`を有効化します。

```
<p class="supports-[display:grid]:grid">
  <!-- ... -->
</p>
```

ちなみに、TailwindCSSをビルドした後のCSSは以下のようになります。

```
@supports (display:grid)
.supports-\[display\:grid\]\:grid {
  display: grid;
}
```

supports-* variantを判定だけに使って、実際のCSSは別のものにしたい場合は以下のようにします。

```
<p class="supports-[display:none]:bg-black">
  <!-- ... -->
</p>
```

ブラウザサポートだけでなく、aria属性でCSSの出し分けもできます[2]。`aria-checked:xx`で`aria-checked=true`のときにスタイルが当たるようになります。他にもこれにより、disabled, pressed, required, selected時などでもスタイル変更が可能になりました。

```
<span
  class="bg-gray-600 aria-checked:bg-blue-600"
  aria-checked="true"
  role="checkbox"
>
  <!-- -->
</span>
```

data属性[3]での出し分けも可能で以下のようにできます。

```
<div data-size="large" class="data-[size=large]:p-8">
  <!-- ... -->
</div>
```

コンテナクエリ

コンテナクエリを使うためのプラグインが追加され、TailwindCSSでコンテナクエリを使えるようになりました[4]。コンテナクエリ自体は主要ブラウザ (Chrome 105以上, Edge 105以上, Safari 16.0以上, Android用Chrome, Safariなど) で使えるようになった新しいCSSです[5]。

メディアクエリは画面のサイズでスタイルを変更させますが、コンテナクエリは親コンポーネントの大きさで子コンポーネントのスタイルを変更できるものになります。少し簡略化してますが、以下のようなコードを書くと、Parentコンポーネントのサイズが600pxのものはデフォルトのままで、300px以下のものはChildコンポーネント内の文字が赤くなります。

```
.parent {
  container-type: inline-size;
}

@container (max-width: 300px) {
  .child {
    color: red
  }
}
```

```
// Childコンポーネント内の文字は変化しない
<Parent class="parent" style="width: 600px" >
  <Child class="child">
</Parent>

// Childコンポーネント内の文字が赤くなる
```

```
<Parent class="parent" style="width: 200px" >
  <Child class="child">
</Parent>
```

これをTailwindCSSで試す際は以下のプラグインをインストールする必要があります。

```
yarn add -D @tailwindcss/container-queries
```

設定ファイルの更新も必要です。

tailwind.config.js

```
theme: {
  // ...
},
plugins: [
  require('@tailwindcss/container-queries'),
  // ...
],
```

使用する際は以下のように使えます。`@container`で指定したものが親となり、`@[xxx]`で親のサイズがいくつになったらスタイルを変えるのかを指定できます。

```
<Parent class="@container">
  <Child class="@[20px]:text-[red]" />
</div>
```

ちなみに、ブラウザサポートの確認を以下のようにできますが、@containerとの組み合わせでは現状動かなかったです。

```
// これは動く
<Parent class="supports-[container]:text-[red]">

// これは動かない
<Parent class="supports-[container]:@container">
```

---

1. https://tailwindcss.com/docs/hover-focus-and-other-states#supports-rules↩
2. https://tailwindcss.com/docs/hover-focus-and-other-states#aria-states↩
3. https://tailwindcss.com/docs/hover-focus-and-other-states#data-attributes↩
4. https://github.com/tailwindlabs/tailwindcss-container-queries↩

5. https://caniuse.com/?search=container↩

whyframe

検証日: 2023年4月30日

whyframeはiframe内のコンテンツをReactやVue, Svelteなどで記述できるようにするライブラリです[1]。whyframeを使わない場合は、以下のように別で作成したアプリをiframe内に読み込ませて使うと思います。

```
<!--
  whyframeを使わない場合の例
  (Reactなどで作成したアプリを別で起動し、読み込ませる)
-->
<iframe src='http://localhost:8000'  ... />
```

whyframeを使うと、同一アプリ内でありながら、iframe内でReactなどが利用できます。まず以下のコードのように、iframeのsrcは静的ファイルにします。

```
<iframe data-why title="Popup 2" src="/frames/special.html">
  <!-- Reactのコード -->
</iframe>
```

静的ファイル内でwhyframeを使うための準備をします。whyframe:appの読み込みはフレームワークにより異なります。Vite, Webpack, Nuxt.js, Next.jsなどで解説されてます[2]。

/frames/special.html

```
<!DOCTYPE html>
<html>
<body>
<div id="app"></div>
</body>

<script type="module">
import { createApp } from 'whyframe:app'
createApp(document.getElementById('app'))
</script>
</html>
```

上記のように設定すると、iframe内でreactなどが使えるようになります。例えば、以下のコードのようにiframe内でuseStateなどを動かすこともできます。ちなみに、iframe内と外でstateを共有してないため、注意が必要です。

index.tsx

```
import { useState } from 'react'

function App() {
  const [count, setCount] = useState(0)

  return (
    <>
      <iframe data-why title="Popup 2" src="/frames/special.html">
        <button onClick={() => setCount(count + 1)}>plus 1</button>
        <p>count: { count }</p>
      </iframe>
    </>
  )
}

export default App
```

---

1. https://whyframe.dev/↩
2. https://whyframe.dev/docs/integrations/vite↩

Memlab

検証日: 2023年5月7日

Memlabはウェブアプリのメモリリークを検出するためのツールです[1]。Memlabはヘッドレスブラウザ (Puppeteer) でページを操作しつつ、ヒープスナップショットを記録、解析することでメモリリークの発生を検出します[2]。動かす際は以下のように実際の操作をコード化しておく必要があります。

```
function url() {
  // 一番最初にアクセスするページ
  return "https://xxxx";
}

async function action(page) {
  // メモリリークを確認するための操作
  await page.click('<クリックする対象>');
}


async function back(page) {
  // ページ離脱の方法を書く
}

module.exports = { action, back, url };
```

ちなみに、操作するページのコードは以下のようにしておきます。ボタンをクリックすると、windowオブジェクトにleakedObjectsというプロパティを生やし、divを1024個追加します。

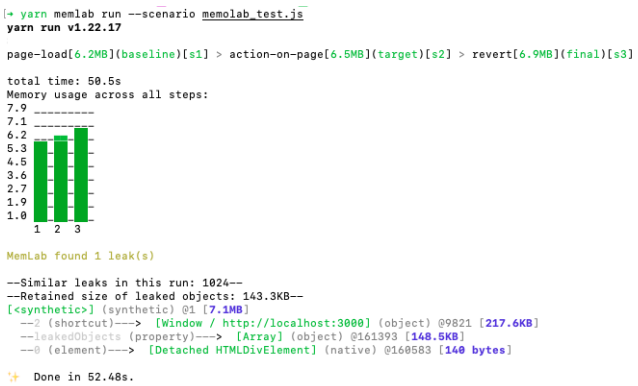
```
import Link from 'next/link';
import React from 'react';

export default function DetachedDom() {
  const addNewItem = () => {
    if (!window.leakedObjects) {
      window.leakedObjects = [];
    }
    for (let i = 0; i < 1024; i++) {
      window.leakedObjects.push(document.createElement('div'));
    }
  };

  return (
    <>
      <button type="button" className="btn" onClick={addNewItem}>
        Create detached DOMs
```

```
      </button>
    </>
  );
}
```

実行結果は以下のようになります。`page-load[6.2MB](baseline)[s1] > action-on-page[6.5MB](target)[s2] > revert[6.9MB](final)[s3]`はそれぞれ、ページアクセス時のヒープサイズ (グラフの1)、ボタンクリック時のヒープサイズ (グラフの2)、ページから離れたときのヒープサイズ (グラフの3) になります。以下の測定結果では、ボタンをクリックしただけなのに、メモリが増えたままになってます。



Memlabの実行結果

グラフの下にある以下の項目で、同じ原因でメモリリークを起こした対象が1024個あり、集計すると143.3KBメモリリークしてることが分かります。

```
--Similar leaks in this run: 1024--
--Retained size of leaked objects: 143.3KB--
```

さらに下にある項目が詳細になります (1024個のうちに1つだけを表示されます)。WindowオブジェクトのleakedObjectsというプロパティ (Array型) に問題があると分かります。さらに見ていくと、Detached HTMLDivElementとあるので、DOMに追加されてないdivが残ったままだということが分かります。

```
[<synthetic>] (synthetic) @1 [7.1MB]
--2 (shortcut)--->[Window / http://localhost:3000] (object) ... [217.6KB]
--leakedObjects (property)--->[Array] (object) ... [148.5KB]
--0 (element)--->[Detached HTMLDivElement] (native) ... [140 bytes]
```

画面操作用のコードを準備する必要がありますが、具体的な原因なども特定できるツールなので、今後より使いやすくなることを期待してます。

---

1. https://engineering.fb.com/2022/09/12/open-source/memlab/↩
2. https://facebook.github.io/memlab/docs/how-memlab-works/↩