

この半年を振り返る

vol.6

# MyWeb ちえんじろぐ

2024年part1



py2wasm

Next.js PPR

Cloudflare SWR

React props.ref

React Aria

Declarative  
Shadow DOM

etc

2023年11月  
~ 2024年5月



# AI編



ここでは、この半年で登場した言語モデルやコーディングアシスト、音声生成モデルについて紹介します。

## 言語モデル

Gemini, Claude3  
Grok-1, ChatGPT-4o  
Llamaについては省略しています。

## コーディングアシスト

Devin  
supermaven  
GitHub Copilot Workspace  
などがあります。waiting list待ちのものも多いです。

## Text-to-Speech

日本語だと  
VoiceVoxやCOEIROINK  
が優秀ですね。  
今回はcoqui-aiを紹介します。

## その他

GitHub Copilot Workspace  
に統合されそうですが、仕様書からコードを生成する  
SpecLangも紹介します。

AI関連技術は毎月毎月良くなっていくため  
すぐに古くなります

# SpecLang

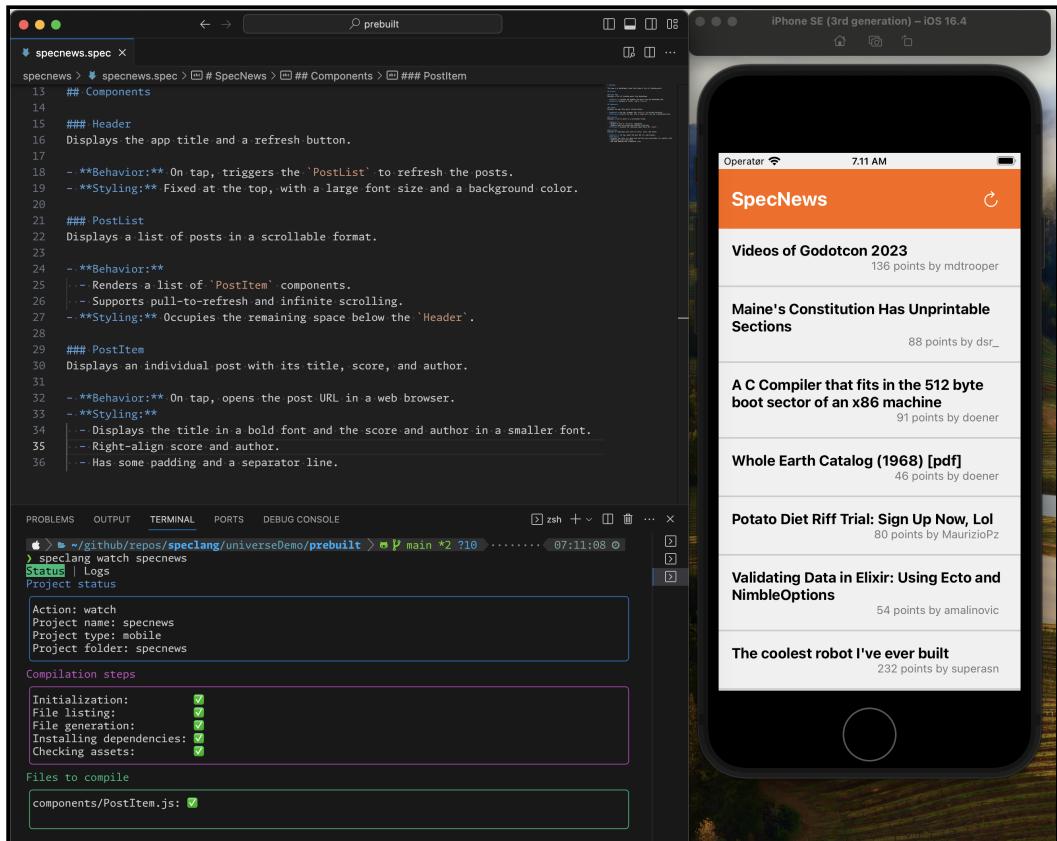
調査日: 2024年3月4日

SpecLangは自然言語で書いた仕様書から、プログラムを生成するためのツールチェーンです<sup>1</sup>。こちらはGitHubが研究を進めており、現状はプロトタイプを作成し、検証している段階です。

具体的な仕組みは不明ですが、言語モデルをそのまま使うのではなく、静的解析や都度ビルドした結果を表示する仕組みを持っています。これにより、仕様書と生成物を見比べながら、実装できるそうです。

ちなみに、仕様を正しく認識して文字に起こす必要があるため、ノーコードツールと異なり、非エンジニア向けのツールではありません。

まだ試すことはできませんが、以下のようにマークダウンで仕様書を記述し、アプリを開発していくそうです。こちらの例では、仕様書からReact Nativeのコードを生成し、アプリとして起動しているそうです。



SpecLangでReact Nativeのコードを生成し、アプリとして起動するデモ

1. <https://githubnext.com/projects/speclang>

# WebAssembly編

WebAssemblyはブラウザでネイティブに動くという範囲を超えて、「一度作れば、既存のものと組み合わせて、どこでもずっと使える」便利なコンパイラ先になりつつあります。

## WASI Preview2

RustやMoonbitなど様々な言語で生成されたWebAssemblyを組み合わせて使えるようにしようぜという仕様です。

## py2wasm

ワンライナーでPythonのコードをWebAssemblyに変換できるツールです。やっぱ、Wasmerはすごい！

## その他

この他には、WebAssemblyのコンパイルができるようになったKotlin/Wasmについても紹介します。

個人的に今一番望んでいるものは、Chromeなど各種ブラウザにSQLiteが入り、Browser APIで簡単に扱えるようになることです。

# py2wasm

検証日: 2024年4月27日

こちらは、Python のコードを WebAssembly に変換するためのツールです<sup>1</sup>。現在は WebAssembly 化された Python のインタープリタを使って Python を動かすことはできますが、Python のコードを WebAssembly に変換するツールはまだ少ないです<sup>2</sup>。Cython で Python WebAssembly をビルドするための開発も続いてますが、まだサポートには時間がかかりそうです<sup>3</sup>。

利用する際は、Python のバージョンを 3.11 にした状態で、以下のようにするだけです。内部でフォークした Nuitka を利用し、Python から C に変換した後、WebAssembly に変換されます。

```
$ py2wasm pystone.py -o pystone.wasm
```

公式ブログで提供されているサンプルコードを使って、通常の Python を使った実行速度と py2wasm で変換した WebAssembly の実行速度、WebAssembly 化した Python インタープリタを使った実行速度を比較すると、このようになります。

py2wasm でビルドした WebAssembly は通常の Python を用いた実行よりも 2 倍ぐらい遅いです。しかし、WebAssembly 化した Python インタープリタ経由での実行速度と比較すると、3 倍近く高速になっています。

通常の Python と比べると遅くなるため、あまり意味はない気がするかもしれませんが、WebAssembly 化したコードは CDN Edge に配置したり、ブラウザや IoT 機器で動かすことができるため、オフラインでの利用やサーバーを経由しない分、高速な処理が可能になる可能性があります。

```
$ python pystone.py
Pystone(1.1) time for 50000 passes = 0.121772
This machine benchmarks at 410604 pystones/second
$
$ wasmer run pystone.wasm
Pystone(1.1) time for 50000 passes = 0.254458
This machine benchmarks at 196496 pystones/second
$
$ wasmer run python/python --mapdir=/app:/app/pystone.py
Pystone(1.1) time for 50000 passes = 0.828336
This machine benchmarks at 60362 pystones/second
```

一番上が通常の Python、真ん中が py2wasm で変換した WebAssembly、一番下が WebAssembly 化した Python インタープリタの実行速度

今後、Cython の WebAssembly 対応が安定になったときに、py2wasm が残るのか、それとも Cython が主流になるのか、注目していきたいところです。個人的には、簡単に試すときは py2wasm で、速度が大事になる時は Cython で WebAssembly 変換が必要になるのかなと想像しています。

1. <https://wasmer.io/posts/py2wasm-a-python-to-wasm-compiler/>

2. うえぶちえんじろぐ 2023part1でSpinについて紹介しました。SpinはPythonのコードをWebAssembly化してサーバーで動かすためのフレームワークです。内部でBytecode Allianceのcomponentize-pyを利用することで、WebAssemblyに変換してますが、複数のファイルが生成されるため、py2wasmのように1ファイルにまとめることはできません。参考URL:  
<https://github.com/bytecodealliance/componentize-py>。 ↪
3. <https://github.com/python/cpython/blob/main/Tools/wasm/README.md> ↪



# ツール編

ここでは、Webアプリ開発を便利にするツールについて紹介します。Dockerは月額課金を始めてから開発サイクルがかなり良くなり、毎回驚かせてくれます。

## Docker Build Cloud

クラウド上でビルトしたイメージを簡単にチームメンバーに共有できるようになりました。画期的！

## Magika

ファイルの種類を識別するためのツールです。かなり精度が高く、ファイル名を変えただけのファイルもキチンと種類を識別できます。

## git replay

replayコマンドというよりも、Git 2.34.0からデフォルトになったマージ戦略のmerge-ortの説明が主です。

## その他

TablePlus社のTinyWebについても簡単に触れてます。

今回あまり量を書いてないです

ただ、Dockerは毎回面白いので、詳し目に実践した結果を書いてます。

# docker init

検証日: 2024年2月9日

この度、docker init コマンドが GA となりました<sup>1</sup>。こちらは、Dockerfile や compose.yaml のテンプレートを生成するためのコマンドです。

利用する環境を選択し、バージョンや利用するポートなどを入力することで、テンプレートが生成されます。ちなみに、「Don't see something you need? Let us know!」という選択肢もあり、サポートして欲しい言語やスタックをリクエストできます。

```
$ docker init

Welcome to the Docker Init CLI!

This utility will walk you through creating the following files with sensible defaults for your
project:
- .dockerignore
- Dockerfile
- compose.yaml
- README.Docker.md

Let's get started!

? What application platform does your project use? [Use arrows to move, type to filter]
Go - suitable for a Go server application
Python - suitable for a Python server application
Node - suitable for a Node server application
Rust - suitable for a Rust server application
ASP.NET Core - suitable for an ASP.NET Core application
PHP with Apache - suitable for a PHP web application
Java - suitable for a Java application that uses Maven and packages as an uber jar
> Other - general purpose starting point for containerizing your application
Don't see something you need? Let us know!
Quit
```

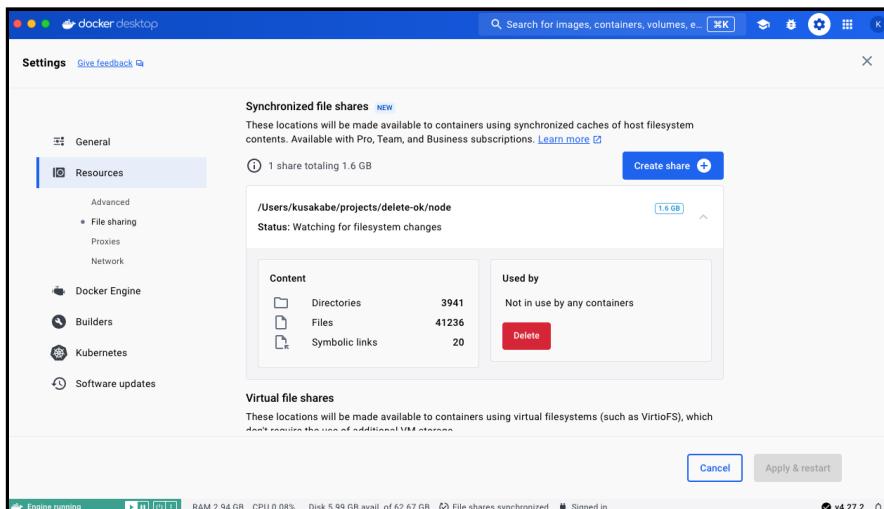
docker init 実行時の画面

# Docker Desktop synchronized file shares

検証日: 2024年2月25日

この機能は Docker の有料プランで利用可能です。ホストのファイルを VM 上にマウント (同期) する際に、ホスト側のファイルシステムキャッシュを活用して、高速なファイル同期を実現する機能です<sup>2</sup>。また、指定したフォルダを監視し、ファイルに変更があった場合は、それが即座に VM 上で同期されます。

これにより、ファイルの所有権周りのトラブルや、ファイルの同期が遅いといった問題を解決できるとのことです。同期は以下の画像のように、Docker Desktop の設定画面からフォルダを選択します。



ファイル同期機能のデモ

実際に、ubuntu のイメージに nodejs のプロジェクトをマウントさせて、検証するために以下のスクリプトを準備しました。これは、あらかじめ clone しておいた nodejs のプロジェクトを ubuntu のイメージにマウントさせ、ファイル数を計算するスクリプトです。

docker\_test.sh

```
#!/bin/bash

total_time=0
PROJECT_DIR=$1

for i in {1..10}; do
    start=$(date +%s.%N)

    # 10回実行されるコード
    docker run --rm -v $PROJECT_DIR:/workspace ubuntu \
    bash -c "find workspace -type f | wc -l" > /dev/null 2>&1

    end=$(date +%s.%N)
    time=$((end - start))
    total_time=$((total_time + time))

done

avg_time=$((total_time / 10))
echo "Average execution time per iteration: $avg_time ms"
```

```
end=$(date +%s.%N)

duration=$(echo "$end - $start" | bc)
total_time=$(echo "$total_time + $duration" | bc)
done

average=$(echo "$total_time / 10" | bc -l)

echo "合計時間: $total_time 秒"
echo "平均時間: $average 秒"
```

結果はこのようになります。synchronized file sharesを有効にしているnodejsのプロジェクトでは、平均0.4秒でファイル数を計算できました。一方有効にしていないnodejsのプロジェクトでは、平均10.8秒となっています。

```
[\$ sh docker_test.sh ./node
合計時間: 4.0 秒
平均時間: .40000000000000000000000000000000 秒
[\$

[\$ sh docker_test.sh ./node2
合計時間: 108.0 秒
平均時間: 10.800000000000000000000000000000 秒
```

synchronized file sharesを有効にした場合と有効にしてない場合での違い

- 
1. <https://www.docker.com/blog/streamline-dockerization-with-docker-init-ga> ↪
  2. <https://docs.docker.com/desktop/synchronized-file-sharing> ↪

# Docker Build Cloud

検証日: 2024年2月26日

こちらはクラウド上で Docker イメージをビルドするサービスです。チームで利用している場合は、他のメンバーとビルドしたイメージを共有して利用できます<sup>1</sup>。

個人でも無料で利用できますが、ビルド時間に制限があったり、他の人のとの共有ができないなど、いくつか制限があります。

利用する際は、Docker Build Cloud 上で Cloud Builder を追加する必要があります。

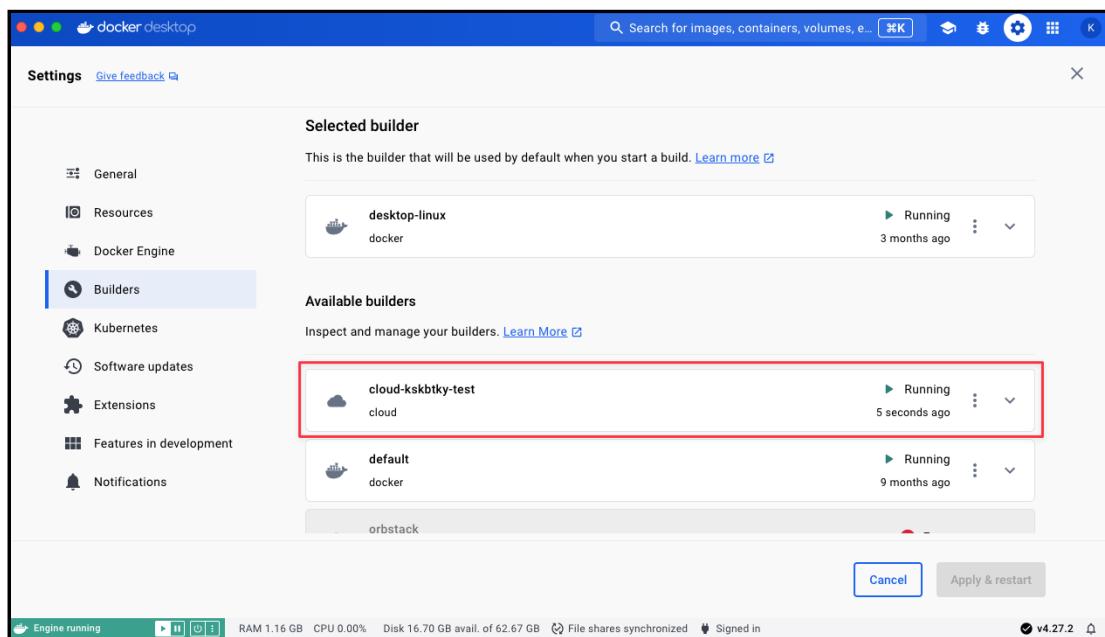
The screenshot shows the Docker Build Cloud interface. On the left, there's a sidebar with a user icon and the name 'kskbtky'. Below it are 'Build Cloud' and 'Settings & billing' buttons. The main area has a header 'kskbtky / Build Cloud'. Underneath, there's a section titled 'Cloud Builders' with the sub-instruction 'Cloud-driven shared builders designed to accelerate build times for your teams.' A link 'Learn more.' is present. A table shows a single builder named 'test' which is 'Enabled'. To the right of the table, there are statistics: 'Remaining build minutes' (98 min), 'Included monthly minutes' (100), 'Total monthly build minutes' (100), 'Build minutes used this cycle' (2), 'Cloud builders' (1 of 1), and 'Remaining reserve minutes' (0). At the bottom right is a button 'Upgrade Docker Build Cloud plan'.

Docker Build Cloud の画面

追加すると、以下のように Docker Desktop 上で Cloud Builder が表示されます。

## Docker Build Cloud

MyWeb ちえんじろぐ 2024part1 (2024/5/22 2:12:50 更新)



Docker Desktop 上で Cloud Builder を追加した画面

この Cloud Builder を利用してビルドする際は以下のようないみになります<sup>2</sup>。cloud-ORG-BUILDER\_NAMEは、Docker Desktopで表示されているCloud Builderの名前になります。

```
docker buildx build --builder cloud-ORG-BUILDER_NAME --tag IMAGE .
```

ビルド後はDocker Desktop上でこのように表示されます。author欄で誰がビルドしたのかが分かります(ローカルでビルドした場合はN/A、チームでDockerを利用している場合は他のメンバーがビルドしたイメージも確認できるそうです)。

Completed builds							
	ID	Name	Builder	Status	Duration	Created	Author
<input type="checkbox"/>	ISHRGO	<a href="#">todolist-node</a>	desktop-linux	✓ Completed	9.9s	2 minutes ago	N/A
<input type="checkbox"/>	9ZMCR8	<a href="#">todolist-node</a>	cloud-kskbtky-test	✓ Completed	3.8s	2 minutes ago	<a href="#">kskbtky</a>

ビルド結果

1. <https://www.docker.com/products/build-cloud> ↪
2. <https://docs.docker.com/build/cloud/usage> ↪



# サービス編

面白サービスや主要ベンダー(Cloudflare, GitHub, AWS)の新機能について紹介しています。主要ベンダーの定義は独断です。

## GitHub

正式リリースされた Markdown Alert の紹介 やマルチアカウントサポートについて紹介します。

## Cloudflare

リリースが予定されている SWR や Cloudflare D1 について紹介します。

## AWS

AWS Lambda のアップデート やインフラのコード化について紹介します。

## その他

Vercel や JSR, Supabase についても紹介します。

GitHub と Cloudflare は毎回面白い発表があるので、紹介します。AWS は会社でメインで使ってるため紹介しますが、Azure とか GCP の話もいつか追加したいです。

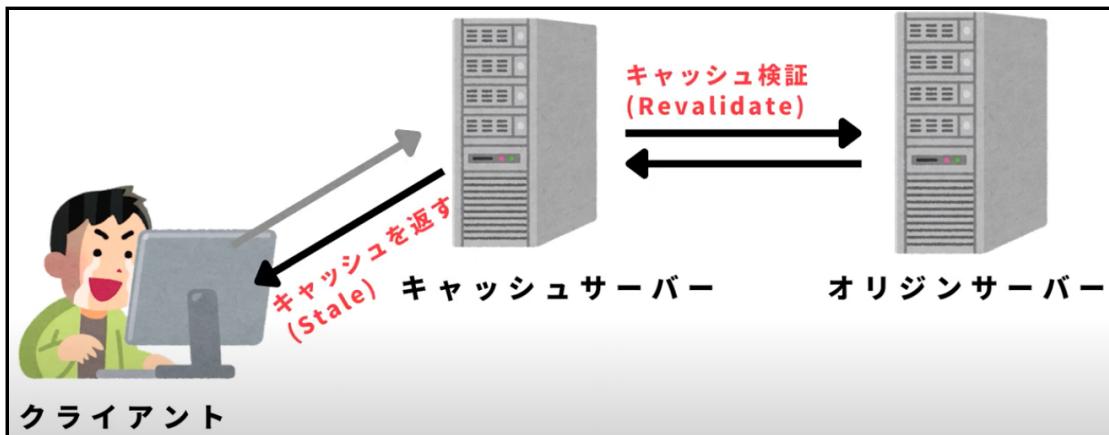
# Cloudflare

## SWR のサポート (予定)

SWR (Stale-While-Revalidate) は RFC 5861 で提唱された HTTP のキャッシュ戦略です。有名なところで言うと、この仕組みを模倣した Vercel の SWR ライブラリなどがあります。

SWR はクライアントからリクエストがあった際に、先にキャッシュを返し (Stale)、その間 (While) にバックエンドでリソースの確認をして、キャッシュの更新をする (Revalidate) 戦略のことです。

2024 年後半から提供予定<sup>1</sup> だそうで、大変楽しみです。Cloudflare から AWS, Vercel などに波及していくので、今後のフロントエンドフレームワークでも SWR をネイティブにサポートしたもののが出てきそうです。



SWR の概念図

## Cloudflare D1 GA

Cloudflare D1 の一般提供が開始されました<sup>2</sup>。Cloudflare D1 は Cloudflare Workers を使ってアクセスできる SQLite ベースの DB です。つまり、CDN Edge 上にグローバル展開されており、ユーザーに物理的に近い位置からアクセスできる DB サービスになります。有名なところで言うと、Next.js の middleware や Shopify の Shopify Hydrogen のバックエンドとして利用されています。

個人的にはグローバル展開していないサービスではあまり必要ない気がします。特に日本だけサービス展開している場合は、過剰な気もしますが、DB サービスにしては圧倒的に安価で無料枠も充実しているので、知っておいて損はないと思います。

	Free プラン	有料プラン (5 ドル/月)
読み取り行数	500 万行/日	250 億行/月 以降は 0.001 ドル/100 万行
書き込み行数	10 万行/日	5,000 万行/月 以降は 1 ドル/100 万行
ストレージ	5GB	5GB 以降は 0.75 ドル/GB, 月

Cloudflare D1の費用について cf. <https://developers.cloudflare.com/d1/platform/pricing/>

ちなみに、Cloudflare Workers 経由でアクセスする場合は以下のようになります。tomlにDBの接続情報を記述し、Cloudflare Workers (index.ts) で、SQLを書いてアクセスします。

wrangler.toml

```
[[d1_databases]]
binding = "DB"
database_name = "test"
database_id = "6ddef1df-xxxx-yyyy-affd-ae39s546129c"
```

/src/index.ts

```
export interface Env {
  DB: D1Database; // wrangler.tomlで指定したbinding名 (DB) を定義
}

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

    if (pathname === "/api") {
      const { results } = await env.DB.prepare("SELECT * FROM Users")
        .all();
      return Response.json(results);
    }

    return new Response("Please Call /api");
  },
};
```

ちなみに、ORMも用意されており、PrismaもCloudflare Workers用にカスタマイズされたものが用意されています<sup>3</sup>。また、REST APIでアクセスすることもでき、以下のようなクエリでDBにアクセスできます<sup>4</sup>。

```
curl --request POST \
  --url https://api.cloudflare.com/client/v4/accounts/account_identifier \
  /d1/database/database_identifier/query \
  --header 'Authorization: Bearer <アクセスキー>' \
  --header 'Content-Type: application/json' \
  --data '{
  "params": [
    "firstParam",
    "secondParam"
  ],
  "sql": "SELECT * FROM myTable WHERE field = ? OR field = ?;"}
```

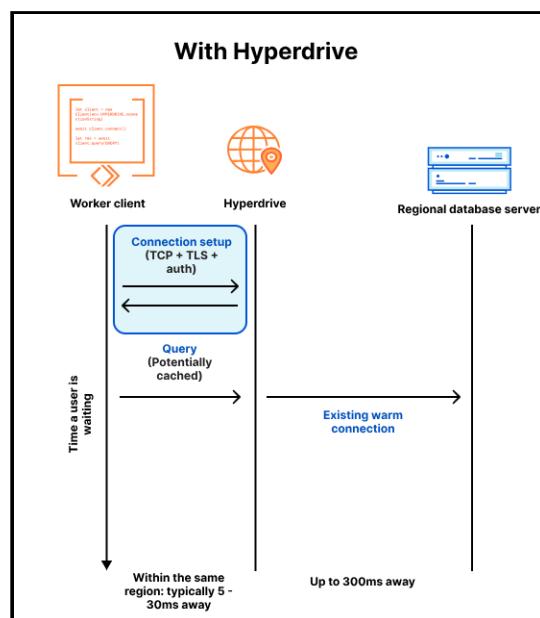
Cloudflare D1 にはこのほかにもデータエクスポート機能やタイムトラベル機能 (特定の期間まで DB を戻せる機能) なども入っています。個人的にはブランチ機能 (DB のバージョンを複数保持できる機能) も今後追加されて欲しいですが、かなり安く使える DB としては魅力的なサービスだと思います。

## Hyperdrive

Hyperdrive は既存の Postgres 互換データベースへ接続するための前段となるコネクションプールを CDN Edge 上でグローバルに管理するためのサービスです<sup>5</sup>。ちなみに、今後は MySQL のサポートもするそうです<sup>6</sup>。

公式の概念図が分かりやすいので、引用します<sup>7</sup>。Hyperdrive では、以下のようにパブリックに公開された DB と Hyperdrive であらかじめコネクションを確立しておきます。クライアントが DB アクセスをするときは、CDN Edge にある Hyperdrive でコネクションを確立した後、キャッシュを返すまたは DB にアクセスするという流れになります。

現状はパブリックに公開された DB に対してしか使えませんが、今後は Magic WAN や Cloudflare Tunnel を用いてプライベート DB にも接続できるようにするそうです<sup>8</sup>。



Hyperdrive の概念図

1. <https://blog.cloudflare.com/browser-rendering-api-ga-rolling-out-cloudflare-snippets-swru-and-bringing-workers-for-platforms-to-our-paygo-plans> ↪
2. <https://blog.cloudflare.com/making-full-stack-easier-d1-ga-hyperdrive-queues-ja-jp> ↪
3. <https://blog.cloudflare.com/prisma-orm-and-d1> ↪
4. <https://developers.cloudflare.com/api/operations/cloudflare-d1-query-database> ↪
5. <https://blog.cloudflare.com/making-full-stack-easier-d1-ga-hyperdrive-queues-ja-jp> ↪
6. <https://developers.cloudflare.com/hyperdrive/get-started/#3-connect-hyperdrive-to-a-database> ↪

7. <https://developers.cloudflare.com/hyperdrive/configuration/how-hyperdrive-works/> ↵
8. <https://developers.cloudflare.com/hyperdrive/examples/google-cloud-sql/#1-allow-hyperdrive-access> ↵

# JSR

検証日: 2024年3月1日

JSR (JavaScript Registry) は Deno が公開した TypeScript や ESM に最適化したレジストリです。Deno でも add コマンドが追加され、簡単に JSR からパッケージを追加できるようになりました<sup>1</sup>。

```
deno add @luca/flag
```

Deno では JSR のサポートを組み込んでいるため、`deno add` すらせずに、以下のようにライブラリの頭に、`jsr:` を指定するだけでも利用できます（ただし、JSR にパッケージがある場合）。

```
import { printProgress } from "jsr:@luca/flag@1";
printProgress();
```

deno.json

```
{
  "imports": {
    "@luca/flag": "jsr:@luca/flag@^1.0.1",
  }
}
```

Node.js で使う場合は npx 経由で追加して、使います。

```
npx jsr add @std/fmt
```

ただし、JSR は ESM 専用のレジストリのため、Node.js で使用する場合は `type: "module"` にする必要があります。

package.json

```
{
  "type": "module",
  "dependencies": {
    "@std/fmt": "npm:@jsr/std_fmt@^0.218.2"
  },
}
```

ちなみに、`npx`で追加した際には `.npmrc` に以下の設定が追加されます。これは `@jsr` のパッケージを `npm.jsr.io` から取得するための設定です。これが先ほど出た `npm:@jsr/std_fmt@^0.218.2` と合わさることで、`npm install` 時にも JSR からパッケージを取得できます。

```
.npmrc
```

```
@jsr:registry=https://npm.jsr.io
```

JSR は npm レジストリ (Node Package Manager) の置き換えを目的にしたものではありません。JSR は以下の観点から、npm レジストリをより発展させたもの (スーパーセット) として開発されました<sup>2</sup>。

- Web プラットフォームでは ESM<sup>3</sup> をメインに使う流れになっている
- JavaScript のランタイムは Node.js だけでなくなった
- TypeScript は事実上の JavaScript のスーパーセットとして標準化している

JSR では TypeScript をネイティブにサポートしているため、ライブラリ提供者は JS にトランスパイルせずにデプロイできます。TypeScript が使えない環境では、npm.jsr.io にて JavaScript にトランスパイルされて配布されます<sup>4</sup>。

また、型情報 (JSDoc) からパッケージのリファレンスドキュメントを自動生成し、オンライン上で確認できます。`@params` や `@returns`、コード例を記載すると、それがドキュメント上でも表示されます。

```
/**  
 * add 関数は 2 つの引数を受け取り、その合計を返します  
 * @param a 1 つ目の数値  
 * @param b 2 つ目の数値  
 * @returns 2 つの数値の合計  
 * ````ts  
 * add(1, 2) // 3  
 * ````  
 */  
export function add(a: number, b: number): number {  
    return a + b  
}
```

The screenshot shows the JSR documentation for the `@pilefort/test` package at version 0.1.0. The main content is the `add` function. Below is a detailed breakdown of the visible information:

- Function:** `function add`
- Implementation Examples:**
  - Deno: `npx jsr add @pilefort/test`
  - npm: `import { add } from "@pilefort/test";`
- Signature:** `add(a: number, b: number): number`
- Description:** add関数は2つの引数を受け取り、その合計を返します
- Parameters:**
  - `a: number`: 1つ目の数値
  - `b: number`: 2つ目の数値
- Return Type:** `number`
- Description:** 2つの数値の合計
- Example:** `add(1, 2) // 3`

## JSRのドキュメントサンプル

その他にも JSR では ESMModule のみをサポートし、Deno だけでなく、Bun や Cloudflare Workers でも動作するようにサポートしているようです。

1. [https://github.com/denoland/deno/pull/22520 ↵](https://github.com/denoland/deno/pull/22520)
2. [https://jsr.io/docs/why ↵](https://jsr.io/docs/why)
3. ESMModule は ES2015 で追加されたモジュールシステム仕様です。Node.jsなどのサーバーサイドで使う JS やブラウザで使う JS で、`import` 文でモジュールを読み込み、`export` 文でモジュールをモジュール外に公開するために使います。対して、CommonJS は Node.js が独自に実装していたライブラリを読み込むためのシステムです。こちらでは `require` 文でモジュールを読み込み、`module.exports` でモジュールをモジュール外に公開します。昔からあるライブラリの読み込み方ですが、TypeScript 化しているプロジェクトでは、ESModule を使うことが多いです。↵
4. (4/27 追記) ちなみに、TypeScript から型定義ファイルの生成には TSC ではなく、Rust 製のツールを使って生成しているようです (SWCかな? と思いますが、ツール名は紹介されてません)。  
[https://deno.com/blog/how-we-built-jsr#turning-typescript-into-js-and-dts-files-for-npm ↵](https://deno.com/blog/how-we-built-jsr#turning-typescript-into-js-and-dts-files-for-npm)



# ライブラリ編



ここでは、React19 Betaの変更点の確認や面白いまたは便利なライブラリの紹介をします。

## React19 Beta

props周りの実装がかなり変わっており、レンダリング速度の向上なども期待できるそうです。

## React Aria

Adobeが開発しているライブラリでW3CのARIA Authoring Practices Guideに沿ったアクセシビリティの高いUIを構築できます。

## ESLint v9.0.0

Flat Configという新しい設定ファイルの書き方がデフォルトになりました。

## その他

ひと昔前に流行ったStyleXについても一般公開されたため紹介しています。

Reactがほぼスタンダードになっているため、React関連ライブラリの話題が多めです。

# React 19 Beta

検証日: 2024年4月30日

React 19 Beta では、Server Component や非同期処理の状態管理の機能追加や React DOM の改善が入っています<sup>1</sup>。

## propsの扱いが変わり、props.refが利用可能に (forwardRef非推奨化)

Reactで入力欄にフォーカスを当てたり、DOM要素を直接扱いたい場合はrefを利用します。以下の例では、子コンポーネントにあるinput要素を親コンポーネントに配置したボタンをクリックすることでフォーカスを当てています。

parent.tsx

```
import { useRef } from 'react';
import MyInput from './MyInput';

export default function Form() {
  const ref = useRef(null);

  const handleClick = () => {
    ref.current.focus();
  }

  return (
    <form>
      <MyInput ref={ref} />
      <button type="button" onClick={handleClick}>Edit</button>
    </form>
  );
}
```

子コンポーネントでrefにアクセスするには、forwardRefを使います。

child.tsx

```
const MyInput = forwardRef((props, ref) => {
  return <input ref={ref} />
});

export default MyInput;
```

React19では、より直感的に以下のように記述できるようになります。また、forwardRefが非推奨になり、将来的に廃止になる予定です。

child.tsx

```
const MyInput = (props) => {
  return <input ref={props.ref} />
};

export default MyInput;
```

なぜ、以前は props に ref が入ってなかったのかというと、React の内部で props を返す前に、key や ref を削除していたためです<sup>2</sup>。

jsx/ReactJSXElement.js

```
const RESERVED_PROPS = {
  key: true,
  ref: true,
  // ...
};

// ...

export function jsx(type, config, maybeKey) {
// ...
  for (propName in config) {
    if (
      hasOwnProperty.call(config, propName) &&
      !RESERVED_PROPS.hasOwnProperty(propName)
    ) {
      // key, refを除いたpropsを生成
      props[propName] = config[propName];
    }
  }
  // ...
}
// ...
```

この度、JSX周りの処理が大幅に変わり、該当部分はこのように改善されました<sup>3</sup>。props に ref が入るようになっただけでなく、config のループ処理が改善されました。これにより、JSX のパフォーマンス自体も改善される可能性があるそうです (PR のタイトルが「Fast JSX: Don't clone props object」になってました)。ただ、クライアントアプリを作る限り、key は指定すると思うため、個人的にはパフォーマンス改善は限定的ではないかなと感じます (フレームワークやライブラリ内で、createElement や cloneElement で key を指定していない場合は、高速かも?)。

shared/ReactFeatureFlags.js

```
// ...
export const enableRefAsProp = true;
export const disableStringRefs = true;
```

jsx/ReactJSXElement.js

```

let props;
if (enableRefAsProp && disableStringRefs && !( 'key' in config )) {
  // configにkeyがない場合、configをそのままpropsに代入
  props = config;
} else {
  props = {};
  for (const propName in config) {
    if (propName !== 'key' && (enableRefAsProp || propName !== 'ref')) {
      if (enableRefAsProp && !disableStringRefs && propName === 'ref') {
        props.ref = coerceStringRef(config[propName], getOwner(), type);
      } else {
        props[propName] = config[propName];
      }
    }
  }
}

```

## Server Component

Server Componentはビルド時に1回だけ実行するか、サーバーを用意してユーザーがリクエストする度に実行できる画面上の一要素(コンポーネント)になります<sup>4</sup>。

「サイドバーなど、どのページでも共通して使いたいけど、ページごとにデータをフェッチしないといけないといった問題」や「ページ上的一部のコンポーネントのデータ取得が遅くて、ページの表示が遅くなるといった問題」などが比較的簡単に解決できる機能です。

Server Componentの説明は以上です。個人的にはReactを生で使う需要よりも、Next.jsやRemixなどのフレームワークに組み込まれたものを使うケースの方が多いと感じるためです。

## 非同期処理の状態管理 (Actions)

Reactのドキュメント上では、非同期な状態遷移を伴う関数のことをActionsと呼んでいます。Actionsはデータ送信や取得を自動的に管理し、今がどのような状態にあるかを追跡できます。

例えば、今まで非同期関数を実行する際、以下のようなコードを書いていました。このコードではページアクセス時に3秒間待機している間はLoading画面を出し、3秒後にデータ取得が完了した旨を画面上に表示、エラー発生時はエラーを表示しています。

```

import { useState, useEffect } from 'react'

// 指定された秒数だけ待機するための関数
const delay = (seconds: number) => {
  return new Promise((resolve) => {
    setTimeout(resolve, seconds * 1000)
  })
}

function DelayExample() {

```

```
const [loading, setLoading] = useState(true)
const [error, setError] = useState("")

useEffect(() => {
  const fetchData = async () => {
    try {
      await delay(3) // 3秒待機する
      setLoading(false)
    } catch {
      setError("エラー発生")
      setLoading(false)
    }
  }

  fetchData()
}, [])

if (loading) return <div>Loading...</div>
if (error) return <div>Error: {error}</div>

return <p>データ取得！</p>
}

export default DelayExample
```

React19ではuseTransition関数を使うことで、このように書くことができます。startTransition内の処理が成功または失敗すると、`loading: false`から`loading: true`に切り替わります。前の処理と比べて明示的に書かない分、不安に感じますが、コードはシンプルになります。

```
import { useState, useEffect, useTransition } from 'react'
// ...

function DelayExample() {
  const [error, setError] = useState("")
  const [loading, startTransition] = useTransition()

  useEffect(() => {
    startTransition(async () => {
      try {
        await delay(3)
      } catch {
        setError("エラー発生")
      }
    })
  }, [])

  if (loading) return <div>Loading...</div>
  if (error) return <div>Error: {error}</div>
```

```
    return <p>データ取得！</p>
}
```

また、データ更新が終わる前に画面上にデータを出しておき、後からエラーになったらエラー画面を出し、成功したら成功後のデータに置き換えるといったことも可能です。以下のように、非同期処理の前に、`useOptimistic` を使うことで、対応できます。

```
import { useState, useEffect, useTransition, useOptimistic } from 'react'

const delay = (seconds: number) => {
  return new Promise((resolve, reject) => {
    setTimeout(reject, seconds * 1000)
  })
}

function DelayExample() {
  const [error, setError] = useState("")
  const [loading, startTransition] = useTransition()
  const [optimisticData, setOptimisticData] = useOptimistic("")

  useEffect(() => {
    startTransition(async () => {
      try {
        setOptimisticData("ロード前に表示するデータ")
        await delay(3)
      } catch {
        setError("エラー発生")
      }
    })
  }, [])

  if (error) return <div>Error: {error}</div>

  return (
    <>
      {loading && <p>{optimisticData} ("Loading...")</p> }
      {!loading && <p>データ取得！</p>}
    </>
  )
}

export default DelayExample
```

## React DOM (Form 要素) のアップデート

`<form>` 要素はサーバーに情報を送信するための HTML<sup>5</sup>ですが、React DOM にも `action` 属性が追加され、非同期処理の管理ができるようになりました。ちなみに、HTMLにおける `<form>` 要素では、`method` 属性が `get` または `post` の場合、指定された URL にページ遷移します。

```
<form action="" method="get">
  <input type="text" name="q" />
  <button type="submit">検索</button>
</form>
```

React DOMでHTMLのような`<form>`要素を再現する際は、`useActionState`を使い、非同期関数とエラーの初期値を設定します。エラーの初期値を設定する理由は、成功時はページ遷移し、失敗時は画面にエラーを表示するためです。

```
const [error, formAction, isPending] = useActionState(updateName, "")
```

`formAction`はReact DOMの`<form>`要素の`action`属性に追加します。`error`や`isPending`は画面切り替えで使用します。

```
return (
  <form action={formAction}>
    <div>
      { isPending && <p>Loading...</p>}
      { !isPending && error }
    </div>
    <input type="text" name="name" />
    <button type="submit" disabled={isPending}>Update</button>
  </form>
)
```

また、`useActionState`の第一引数の非同期関数(`updateName`)は、引数に前回の状態とフォームデータを受け取ります。フォームデータは、`formData.get("<name属性の値>")`で値を取得できます。

```
async function updateName(_previousState: string, formData: FormData) {
  const newData = formData.get("name")?.toString() || ""
  // ...
}
```

全体像はこのようになります。

```
import { useActionState } from "react"

function StatefulForm() {
  const [error, formAction, isPending] = useActionState(updateName, "")

  async function updateName(_previousState: string, formData: FormData) {
```

```
const newData = formData.get("name")?.toString() || ""

try {
  // 非同期処理
  // redirect処理
} catch(e) {
  return String(e)
}

return (
  <form action={formAction}>
    <div>
      { isPending && <p>Loading...</p>}
      { !isPending && error }
    </div>
    <input type="text" name="name" />
    <button type="submit" disabled={isPending}>Update</button>
  </form>
)
}

export default StatefulForm
```

## その他

この他にも、Promiseが解消されるまでレンダリングしないuse APIやリソースのプリロードをするためのAPIなども追加されています。メタタグ周りもReactでビルトインサポートしているらしいので、詳細はドキュメントを参照してください。

- 
1. <https://react.dev/blog/2024/04/25/react-19> ↵
  2. <https://github.com/facebook/react/blob/v18.3.1/packages/react/src/jsx/ReactJSXElement.js#L247> ↵
  3. <https://react.dev/reference/rsc/server-components> ↵
  4. <https://developer.mozilla.org/ja/docs/Web/HTML/Element/form> ↵

## ESLint v9.0.0

v9.0.0において、Flat Configがデフォルトになりました。Flat Configは複雑になりがちな ESLint の設定を1つのファイルで簡潔に定義するための仕組みです<sup>1</sup>。既存のプラグインは引き続き利用できますが、設定ファイルの書き方が大きく変わります。

例えば、以前は以下のような設定ファイルを用意していました。

.eslintrc.json

```
{  
  "extends": "eslint:recommended",  
  "parserOptions": { "ecmaVersion": 12, "sourceType": "module" },  
  "rules": { "no-unused-vars": "warn", "no-undef": "warn" }  
}
```

コマンド実行時はextオプションをつけ、検査対象のファイル拡張子をしていします。

```
yarn eslint --ext .js .
```

v9.0.0からはextオプションは廃止となります。代わりに以下のように、設定ファイル内にどの拡張子を検査するかを記述します。また、プラグインもimportして利用します。読み替えは戸惑いますが、プラグイン周りは明示的に書くため、分かりやすくなった気がします。

eslint.config.js

```
import js from "@eslint/js";  
  
export default [  
  js.configs.recommended,  
  {  
    files: ["**/*.js"],  
    languageOptions: {  
      parserOptions: { ecmaVersion: 12, sourceType: "module" },  
    },  
    rules: { "no-unused-vars": "warn", "no-undef": "warn" }  
  }  
];
```

こちらはすぐに置き換えたくなるところですが、Next.jsなどの一部フレームワークでは ESLint と統合されているため、フレームワーク側のバージョンアップと合わせる必要があります。

1. <https://eslint.org/blog/2022/08/new-config-system-part-2/>



# 番外編

ここでは、分類に失敗したものをまとめています。ブラウザ仕様やHTML/CSS/JS関連、プラットフォーム関連、ちょっとした小話が多いです。

## Declarative Shadow DOM

Web Componentsの文脈で登場する概念です。  
仮想DOMなんて使わず、ブラウザのネイティブ機能でアプリ作ろうぜというもの。

## JavaScript

最近はブラウザ実装が先行し、後からECMAScriptに入る事例も多いです。

## Compression dictionary transport

次世代のコンテンツ圧縮方法です。実装が大変そうですが、コンテンツ更新が激しいサイトに有効

## その他

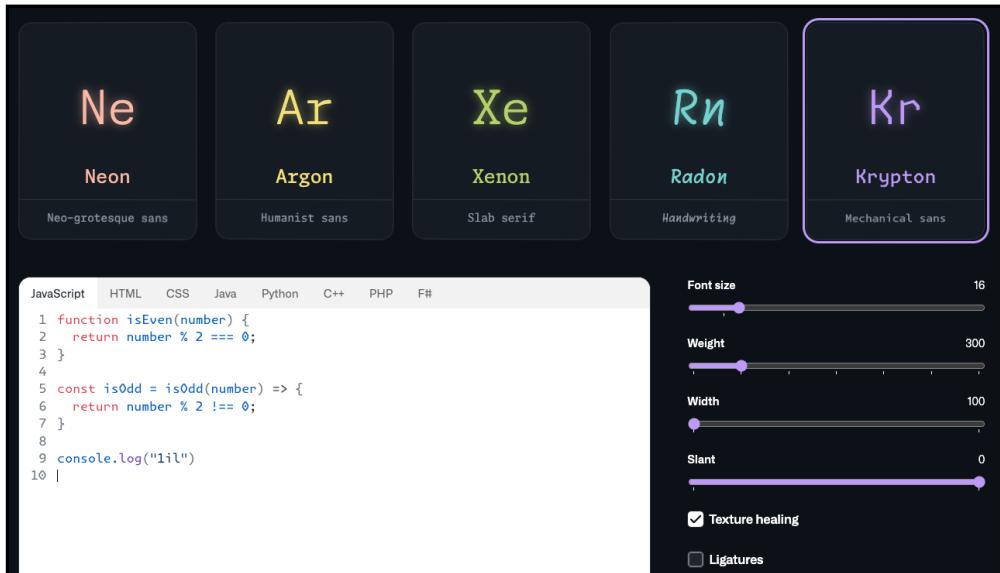
GitHubが開発しているソースコード用のフォントmonospaceの紹介もあります。

フロントは基本的にReact (Next.js) の後追い状態ですが、Web Componentsはブラウザのネイティブ機能を使うため、ゲームチェンジャーになりそうに感じてます。

# monospace (コード用フォント)

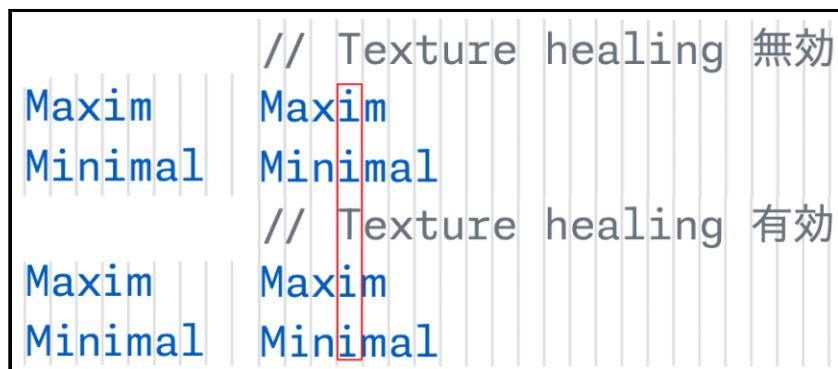
調査日: 2023年12月14日

monospace はコード用フォントで、SIL Open Font License 1.1 (私的利用 OK、商用利用 OK) で提供されています。フォントスタイルは Neon , Argon , Xenon , Radon , Krypton の 5種類があり、それぞれ Texture healing と Ligatures を有効・無効化できます<sup>1</sup>。



monospace

Texture healing は隣り合う文字によって、文字の大きさを調整する機能です。iやlなどの細い文字とmやwなどの太い文字が隣り合った時に、iを少し細くし、mを少し太くすることで、単語全体の幅を変えずに、文字間のバランスを取ることができます。若干文字間の窮屈感が減って、読みやすくなったり気がします。



Texture healing

Ligaturesは、複数の文字が連なった時に、それを1つの文字に置き換える機能です。例えば != が ≠ になったり、 -> が → になったりします。これは好き好きがあると思いますが、有効・無効の切り替えもできるそうです。

! ~	≠
==	=
=>	⇒
!=	≠
***	→
~>	↝
<~	↜
</>	⟨⟩
>	▷

Ligatures (左が無効、右が有効にした場合の見た目)

- 
1. <https://monaspace.githubnext.com> ↵

# Browser API

## CloseWatcher API

検証日: 2024年3月11日

CloseWatcher APIはWICGで提案を出されており、ChromeやEdgeで既に利用可能なAPIです。こちらはプラットフォームごとに異なる閉じるイベントを統一的に扱うためのものです<sup>1</sup>。

こちらは、PCであれば、ESCボタンを押したときに発火し、スマホであれば、戻るボタンを押したときに発火します。今までのウェブサイトの場合、モーダル表示中に戻るボタンを押すと、前の画面に戻ってしまいます。こちらのAPIを使うことで、前の画面に戻らず、モーダルを閉じることができます。

index.html

```
<button id="open">モーダルを開く！</button>
<div id="sidebar" style="display: none">モーダル表示！</div>

<script type="module">
const openElement = document.querySelector("#open")
const sidebarElement = document.querySelector("#sidebar")

openElement.addEventListener("click", () => {
  sidebarElement.style = "display: block"

  const watcher = new CloseWatcher()

  watcher.addEventListener("close", () => {
    sidebarElement.style = "display: none"
  })
})
</script>
```

## Custom Highlight API

こちらは指定した範囲内のテキストにハイライトを当てるためのAPIです<sup>2</sup>。以下のように指定すると、pタグ内の5文字目以降から24文字目までの範囲にハイライトを当てることができます。こちらもほぼ全てのブラウザで利用可能ですが、扱いが少々難しいです。

```
<p>これは、Custom Highlight APIのサンプルです！</p>

<style>
:highlight(custom-highlight-sample) {
  background-color: #93c47d;
  color: black;
}
```

```
</style>

<script>
  const paragraph = document.querySelector('p');
  const range = new Range();
  range.setStart(paragraph.firstChild, 4);
  range.setEnd(paragraph.firstChild, 24);

  const customHighlight = new Highlight(range);
  CSS.highlights.set('custom-highlight-sample', customHighlight);
</script>
```

これは、Custom Highlight APIのサンプルです!

Custom Highlight API のデモ

## Declarative Shadow DOM

Shadow DOM とは、Web Components の文脈で登場する概念です。Web Components とは、ブラウザのネイティブ機能を使って、独自の HTML 要素を再利用可能なものとして使うための機能です<sup>3</sup>。中でも Shadow DOM は、メインとなる DOM から分離してレンダリングすることで、マークアップ (HTML/CSS) やスタイル、スクリプトをメインのものと分離してレンダリングさせることができます (CSS などの衝突を無視できます)。

例えば、以下のように書くことで、`<profile-card>` 要素を既存の要素とは独立して作成してレンダリングできます。

```
<profile-card name="John Doe" email="john@example.com"></profile-card>

<script>
  // <profile-card> 要素の定義
  class ProfileCard extends HTMLElement {
    constructor() {
      super();

      // プロフィールカードのテンプレートを定義
      const template = document.createElement('template');
      template.innerHTML =
        '<div class="profile-card">
          <div class="info">
            <h2 class="name"></h2>
            <p class="email"></p>
          </div>
        </div>
      <style>
        .profile-card { display: flex; align-items: center; }
        .info { flex: 1; }
```

```

    .name { margin: 0; font-size: 18px; }
    .email { margin: 5px 0 0; font-size: 14px; color: #666; }
  </style>
  ;

  // Shadow DOMの作成
  const shadow = this.attachShadow({ mode: 'open' });

  // テンプレートの内容をShadow DOMに追加
  shadow.appendChild(template.content.cloneNode(true));

  // 属性値を要素に反映
  const name = shadow.querySelector('.name');
  const email = shadow.querySelector('.email');

  name.textContent = this.getAttribute('name');
  email.textContent = this.getAttribute('email');
}

// カスタム要素の登録
customElements.define('profile-card', ProfileCard);
</script>

```



Web Componentsでレンダリングされた要素のHTML

Web Components は上記のように、JavaScript API を使って独自要素を作成するため、今まででは SSR などのサーバー上で HTML 要素を作成する場合には利用できませんでした。しかしこの度、Declarative Shadow DOM が登場し、以下のように記述できるようになりました (全てのブラウザで利用可能)。

```

<profile-card>
  <template shadowrootmode="open">
    <div class="profile-card">
      <div class="info">
        <h2 class="name"><slot name="name"></slot></h2>

```

```
<p class="email"><slot name="email"></slot></p>
</div>
</div>
<style>
  .profile-card { display: flex; align-items: center; }
  .info { flex: 1; }
  .name { margin: 0; font-size: 18px; }
  .email { margin: 5px 0 0; font-size: 14px; color: #666; }
</style>
<script></script>
</template>
<span slot="name">John Doe</span>
<span slot="email">john@example.com</span>
</profile-card>

<script>
  class ProfileCard extends HTMLElement {
    connectedCallback() {
      // サーバーからデータを取得
      setTimeout(() => {
        this.fetchData();
      }, 3_000);
    }

    async fetchData() {
      const data = { name: '名無しの権兵衛', email: 'nanashi@email.com' }

      this.querySelector('span[slot="name"]').textContent = data.name;
      this.querySelector('span[slot="email"]').textContent = data.email;
    }
  }

  customElements.define('profile-card', ProfileCard);
</script>
```

JavaScript API 上で Shadow DOM を作成して、要素を追加する処理 (attachShadow など) が `<template shadowrootmode="open">` で置き換えできるようになりました。また、slot 要素を使うことで、Shadow DOM 内に挿入する要素を差し替えできます (デフォルト値を表示しつつ、サーバーのレスポンスによって値を差し替えたりできる)。

Web Components は React や Vue などの仮想 DOM を利用するライブラリとも、Svelte のような DOM を直接生成するライブラリとも独立しているため、部分的な処理改善で今後活躍するかもしれません。ただ、Web Components をそのまま使うのは少々難しいため、Lit などの Web Components をベースとしたライブラリを使うと良いかもしれません。

## iframe の lazy loading

FireFox 121 のリリースにより、全てのブラウザで iframe の lazy loading がサポートされるようになりました。lazy loading を使うことで、画面外にある iframe の読み込みを遅延させることができます<sup>4</sup>。

```
<iframe src="xxxxx" loading="lazy"></iframe>
```

- 
1. <https://github.com/WICG/close-watcher> ↵
  2. [https://developer.mozilla.org/en-US/docs/Web/API/CSS\\_Custom\\_Highlight\\_API](https://developer.mozilla.org/en-US/docs/Web/API/CSS_Custom_Highlight_API) ↵
  3. [https://developer.mozilla.org/en-US/docs/Web/API/Web\\_components](https://developer.mozilla.org/en-US/docs/Web/API/Web_components) ↵
  4. <https://www.mozilla.org/en-US/firefox/121.0/releasenotes> ↵