

haroin57-web コードベースドキュメント

本ドキュメントでは、haroin57-webプロジェクトの全ファイル、関数、クラス、実装、依存関係を解説します。

目次

1. プロジェクト構成
2. エントリーポイント
3. コンテキスト
4. コンポーネント
5. ルート（ページ）
6. 管理者機能
7. ライブラリ・ユーティリティ
8. データファイル
9. 依存関係一覧
10. セキュリティ実装
11. `pv-worker.ts` 詳細解説
12. ローカル記事のデプロイ

プロジェクト構成

```
src/
├── main.tsx          # クライアントエントリ (hydrate対応)
├── entry-server.tsx  # SSG/SSR用エントリ (renderToString)
├── App.tsx           # ランディングページ (トップページ)
├── index.css          # グローバルスタイル
└── contexts/
    └── AdminAuthContext.tsx # Firebase認証コンテキスト
└── components/
    ├── AnimatedRoutes.tsx   # ルーティング (コード分割 + preload/prefetch)
    ├── ServerRoutes.tsx     # SSG/SSR向けRoutes (同期import)
    ├── GlobalBackground.tsx # 背景画像 + p5背景アニメ管理
    ├── P5HypercubeBackground.tsx # p5.js: 4次元立方体背景
    ├── ScrollTopHomeSwitch.tsx # スクロール/スワイプによるページ切り替え
    ├── ClientOnly.tsx       # SSG/SSRでクライアントのみ描画
    ├── AccessCounter.tsx    # PVカウンター
    ├── SiteFooter.tsx       # フッター (PVカウンター含む)
    ├── PrefetchLink.tsx     # プリフェッチ対応リンク
    ├── Lightbox.tsx         # 画像モーダル
    ├── MermaidRenderer.tsx  # Mermaidダイアグラム描画
    ├── BackButton.tsx       # 戻るボタン
    └── admin/
        └── MarkdownEditor.tsx # Markdownエディタ (画像アップロード対応)
└── routes/
    ├── Home.tsx          # ホームページ (コンテンツ一覧)
    ├── Posts.tsx          # 記事一覧ページ
    ├── PostDetail.tsx      # 記事詳細ページ
    ├── Products.tsx        # プロダクト一覧ページ
    ├── ProductDetail.tsx   # プロダクト詳細ページ
    ├── Photos.tsx          # 写真ギャラリーページ
    ├── BBSList.tsx         # BBS スレッド一覧ページ
    ├── BBSThread.tsx        # BBS スレッド詳細ページ
    └── admin/
        ├── PostEditor.tsx   # 記事編集ページ
        └── ProductEditor.tsx # プロダクト編集ページ
└── lib/
    ├── firebase.ts          # Firebase初期化
    └── draftStorage.ts      # 下書き保存ユーティリティ
└── types/
    └── p5.d.ts             # p5モジュール型宣言
└── data/
    ├── posts.json          # 記事データ (ビルド時生成)
    ├── products.json        # プロダクトデータ
    ├── product-posts.json   # プロダクト詳細記事
    └── photos.ts            # 写真データ定義
```

※ SSG (静的HTML生成) は `src/entry-server.tsx` と `scripts/prerender.ts` を利用します。

エントリーポイント

src/main.tsx

アプリケーションのエントリーポイント。

```
const rootElement = document.getElementById('root')
if (!rootElement) throw new Error('Root element not found')

const app = (
  <BrowserRouter>
    <AdminAuthProvider>
      <GlobalBackground />
      <ScrollTopHomeSwitch />
      <AnimatedRoutes />
    </AdminAuthProvider>
  </BrowserRouter>
)

if (rootElement.hasChildNodes()) {
  hydrateRoot(rootElement, app) // SSG/SSRで事前生成されたHTMLを利用
} else {
  createRoot(rootElement).render(app)
}
```

依存関係:

- react-dom/client : createRoot, hydrateRoot
- react-router-dom : BrowserRouter
- ./components/AnimatedRoutes
- ./components/GlobalBackground
- ./components/ScrollTopHomeSwitch
- ./contexts/AdminAuthContext

src/entry-server.tsx

SSG/SSR（静的HTML生成）用のサーバーエントリ。vite build --ssr で dist/server/ に出力され、scripts/prerender.ts から呼び出されます。

```

import { renderToString } from 'react-dom/server'
import { StaticRouter } from 'react-router'

export function render(url: string) {
  return renderToString(
    <StaticRouter location={url}>
      {/* ... */}
    </StaticRouter>
  )
}

```

依存関係:

- react-dom/server : renderToString
- react-router : StaticRouter

scripts/prerender.ts

SSG（静的HTML生成）スクリプト。dist/index.html をテンプレートにして、dist/server/entry-server.* の render(url) を呼び出し、各ルートのHTMLを dist/<route>/index.html として書き出します。

主な挙動:

- 対象ルート: ['/', '/home', '/posts', '/products', '/photos']
- 記事詳細: src/data/posts.json の slug から /posts/:slug を生成
- プロダクト詳細: src/data/products.json の slug から /products/:slug を生成
- BBS や管理者ページなどはプリレンダ対象外 (SPAとして動作)

関連するビルドフロー (package.json) :

- vite build (クライアント)
- vite build --ssr src/entry-server.tsx --outDir dist/server (SSRバンドル)
- node --experimental-strip-types scripts/prerender.ts (HTML書き出し)

src/App.tsx

ランディングページ（ルートパス /）のコンポーネント。

関数:

関数名	説明
App()	ランディングページをレンダリング。ヒーローセクションと「進む」ボタンを表示

関数名	説明
handleNavigate()	/home ヘナビゲート

機能:

- reveal要素のアニメーション表示 (queueMicrotaskで即座にクラス追加)
- ページ読み込み時にスクロール位置をトップにリセット
- ホバーアニメーション付きの進むボタン

依存関係:

- react : useEffect , useRef , useCallback
- react-router-dom : useNavigate

コンテキスト

src/context/AdminAuthContext.tsx

Firebase認証とセッション管理を提供するReactコンテキスト。

定数:

定数名	値	説明
SESSION_TIMEOUT_MS	60 * 60 * 1000	セッションタイムアウト (1時間)
ADMIN_EMAILS	環境変数から取得	管理者メールアドレスリスト

型定義:

```

type BeforeLogoutCallback = () => void | Promise<void>

type AdminAuthContextType = {
  isAdmin: boolean           // 管理者かどうか
  user: User | null          // Firebaseユーザー
  idToken: string | null     // Firebase IDトークン
  isLoading: boolean          // 認証状態読み込み中
  sessionExpiresAt: number | null // セッション期限
  loginWithGoogle: () => Promise<boolean> // Googleログイン
  logout: () => Promise<void>                 // ログアウト
  registerBeforeLogout: (callback: BeforeLogoutCallback) => () => void // ログアウト前コールバック登録
}

```

関数:

関数名	説明
AdminAuthProvider({ children })	認証コンテキストプロバイダー
useAdminAuth()	認証コンテキストを取得するフック
loginWithGoogle()	Googleポップアップでログイン。管理者でなければ即サインアウト
logout(skipCallbacks?)	ログアウト処理。登録済みコールバック（下書き保存など）を先に実行
startSessionTimeout()	1時間のセッションタイムアウトを設定
registerBeforeLogout(callback)	ログアウト前に実行するコールバックを登録。登録解除関数を返す

機能:

- Firebase認証状態の監視（onAuthStateChanged）
- IDトークンの自動更新（50分ごと）
- 1時間のセッションタイムアウト
- ページリロード時のセッション維持
- ログアウト前コールバックシステム（下書き自動保存用）

依存関係:

- react : createContext, useContext, useState, useCallback, useEffect, useRef
- firebase/auth : signInWithPopup, signOut, onAuthStateChanged, User
- ../lib/firebase : auth, googleProvider

コンポーネント

src/components/AnimatedRoutes.tsx

ルーティング（コード分割 + preload/prefetch）を管理。

ルートローダー（例）：

```
const loadHome = preload(() => import('../routes/Home'))           // 高優先度
const loadBBSList = prefetch(() => import('../routes/BBSList'))   // 低優先度
const loadPostDetail = lazyLoad(() => import('../routes/PostDetail')) // ユーザー操作時
```

機能:

- React.lazy + Suspenseによる全ルートの遅延読み込み（動的import）
- preload / prefetch / lazyLoad でルートの読み込み優先度を制御
- 回線状況に応じて、遷移時に関連ルートを順番にプリロード（shouldPrefetch()）

依存関係:

- react-router-dom : Routes , Route , useLocation
- react : lazy , Suspense , useEffect
- ./lib/network : shouldPrefetch
- ./lib/preload : preload , prefetch , lazyLoad

src/components/ServerRoutes.tsx

SSG/SSR向けのRoutes定義（同期import）。src/entry-server.tsx から利用します（管理者ルートは含めません）。

機能:

- 各ページを同期importして即時レンダリング
- AnimatedRoutes.tsx（クライアント向けの遅延ロード）と責務分離

依存関係:

- react-router-dom : Routes , Route , useLocation

src/components/GlobalBackground.tsx

レスポンシブ背景画像 + p5背景アニメを管理。

定数:

```
const BACKGROUND_SRC = '/background-1920.webp'
const BACKGROUND_SRCSET = [
  '/background-1280.webp 1280w',
  '/background-1920.webp 1920w',
  '/background-2560.webp 2560w',
  '/background-3840.webp 3840w',
].join(', ')
```

機能:

- P5HypercubeBackgroundによる背景アニメ描画
- srcsetによるレスポンシブ画像読み込み
- パスに応じた透明度変更（/では1、それ以外では0.45）
- CSS変数による動的ブラー・スケール適用（--bg-blur , --bg-scale）
 - 記事/プロダクト詳細ページで設定されるブラーが、背景画像とp5側の両方に反映される

依存関係:

- react-router-dom : useLocation
- ./P5HypercubeBackground

src/components/P5HypercubeBackground.tsx

p5.js (WEBGL) で「4次元立方体 (tesseract)」の投影アニメーションを背景として描画。

機能:

- p5 を動的importして初期バンドルを軽量化
- 4D回転 → 3D投影 → 2D投影でワイヤーフレームを描画
- パフォーマンス調整 (低解像度レンダリング、FPS制限、 pixelDensity(1) 、 noSmooth())
- prefers-reduced-motion では描画停止

依存関係:

- react : useEffect , useRef
- p5 (dynamic import)

src/components/ScrollTopHomeSwitch.tsx

スクロール/スワイプジェスチャーで / と /home 間を切り替え。

定数:

定数名	値	説明
TOP_PATH	'/'	トップページパス
HOME_PATH	'/home'	ホームページパス

関数:

関数名	説明
normalizeWheelDeltaY(event)	ホイールイベントのdeltaYを正規化
shouldIgnoreTarget(target)	入力要素等を無視するか判定

機能:

- ホイールスクロールで累積値120px以上で遷移
- タッチスワイプで72px以上で遷移
- 900msのクールダウン
- 入力要素やカスタム属性付き要素は無視

依存関係:

- react : useEffect , useRef

- react-router-dom : useLocation , useNavigate

src/components/ClientOnly.tsx

SSG/SSRでのhydration差分や window 依存の副作用を避けるため、マウント後にのみ子要素を描画するラッパー。

機能:

- 初回レンダリングでは null を返す
- useEffect でマウントを検知して描画を開始

依存関係:

- react : useEffect , useState

src/components/SiteFooter.tsx

サイト共通のフッター。 AccessCounter (PVカウンター) は ClientOnly 内で描画し、 SSG/SSRで静的HTMLに含めないようにします。

依存関係:

- ./AccessCounter
- ./ClientOnly
- ../styles/typography : MAIN_TEXT_STYLE

src/components/AccessCounter.tsx

PV (ページビュー) カウンターを表示。

定数:

定数名	値	説明
API_ENDPOINT	'/api/pv'	PVカウントAPI
CACHE_KEY	'haroin-pv-last'	localStorageキャッシュキー

機能:

- 初回レンダリング時にAPIへPOSTリクエスト
- localStorageにカウントをキャッシュ
- オフライン時はキャッシュ値を表示

依存関係:

- react : useEffect , useRef , useState

src/components/PrefetchLink.tsx

ホバー/フォーカス時にルートチャunkをプリフェッチするリンクコンポーネント。

型定義:

```
interface PrefetchLinkProps extends LinkProps {  
  enablePrefetch?: boolean // プリフェッチを有効にするか (デフォルト: true)  
}
```

関数:

関数名	説明
normalizePathname(raw)	パスからクエリ・ハッシュを除去
prefetchRouteChunk(rawPath)	パスに対応するチャunkをプリフェッチ

機能:

- ホバー・フォーカスでルートチャunkを動的インポート
- 一度プリフェッチしたらスキップ

依存関係:

- react-router-dom : Link , LinkProps
- react : useCallback , useRef

src/components/Lightbox.tsx

画像モーダル（ライトボックス）コンポーネント。

Props:

```

type LightboxProps = {
  isOpen: boolean           // モーダル表示状態
  onClose: () => void      // 閉じるコールバック
  imageSrc: string           // 画像URL
  imageAlt: string            // 画像alt
  children?: React.ReactNode // 追加コンテンツ
}

```

機能:

- Escapeキーで閉じる
- 背景クリックで閉じる
- 閉じるボタン
- 画像下に追加情報を表示可能

依存関係:

- react : useEffect

src/components/MermaidRenderer.tsx

Mermaidダイアグラムをレンダリング。

コンポーネント:

コンポーネント名	Props	説明
MermaidRenderer	{ chart, className }	Mermaidコードをレンダリング
MermaidBlock	{ code }	Markdownから抽出されたMermaidブロック用ラッパー

機能:

- mermaidライブラリを使用した非同期レンダリング
- ダークテーマ設定
- エラー時はエラーメッセージと元コードを表示

依存関係:

- react : useEffect , useRef , useState
- mermaid

src/components/BackButton.tsx

戻るボタンコンポーネント。

Props:

```
type BackButtonProps = {
  to?: string // 遷移先 (デフォルト: '/home')
}
```

機能:

- 指定パスへナビゲート
- ホバーアニメーション

依存関係:

- react : useCallback
- react-router-dom : useNavigate

src/components/admin/MarkdownEditor.tsx

管理者用Markdownエディタ。

型定義:

```
type FrontmatterData = {
  title?: string
  summary?: string
  date?: string
  tags?: string[]
}

type MarkdownEditorProps = {
  value: string
  onChange: (value: string) => void
  height?: number
  placeholder?: string
}
```

関数:

関数名	説明
parseFrontmatter(markdown)	Markdownからfrontmatterをパース
FrontmatterHeader({ data })	Frontmatter情報をヘッダーとして表示
MermaidRenderer({ code })	プレビュー用Mermaidレンダラー
extractTextFromChildren(children)	ReactNodeからテキストを抽出

関数名	説明
CodeBlock({ className, children })	カスタムコードブロック (Mermaid対応)
PreviewWithFrontmatter({ source })	Frontmatter付きプレビュー
handleImageUpload(file)	画像をCMS APIへアップロード
handleDrop(e)	ドラッグ&ドロップで画像アップロード
handlePaste(e)	ペーストで画像アップロード

機能:

- @uiw/react-md-editor ベースのエディタ
- ライブプレビュー
- Frontmatterのペースと表示
- Mermaidダイアグラムのプレビュー
- KaTeX式のプレビュー
- 画像のドラッグ&ドロップ/ペーストアップロード

依存関係:

- react : useState , useCallback , useRef , useEffect , useMemo
- @uiw/react-md-editor
- rehype-katex , remark-math
- mermaid
- ../../contexts/AdminAuthContext

ルート (ページ)

src/routes/Home.tsx

ホームページ (コンテンツ一覧)。

型定義:

```
type Interest = { title: string; text: string }
type PostMeta = { slug?: string; title?: string; createdAt?: string }
```

機能:

- Interests (興味分野) の折りたたみ表示
- 最新記事5件の表示
- Products, Photos, BBSへのリンク
- フッター (AccessCounter、SNSリンク)

依存関係:

- react : useEffect , useRef , useState , useCallback
- react-router-dom : useNavigate
- ../components/PrefetchLink , ../components/AccessCounter
- ../data/posts.json

src/routes/Posts.tsx

記事一覧ページ。

型定義:

```
type Post = {
  slug?: string
  title?: string
  html?: string
  summary?: string
  createdAt?: string
  tags?: string[]
}
```

機能:

- CMS APIから記事一覧を取得（フォールバック: 静的JSON）
- タグによるフィルタリング
- 管理者UI（新規作成、編集、ログアウト）

依存関係:

- react : useSearchParams , useEffect , useMemo , useRef , useCallback , startTransition , useState
- react-router-dom : Link
- ../data/posts.json
- ../components/AccessCounter , ../components/PrefetchLink
- ../contexts/AdminAuthContext

src/routes/PostDetail.tsx

記事詳細ページ。

機能:

- Mermaidダイアグラムのレンダリング
- コードブロックのコピーボタン

- スクロールに応じた背景ブラー効果
- Good (いいね) 機能
- X (Twitter) シェアボタン
- OGP/Twitterカードのメタタグ設定

関数:

関数名	説明
extractCodeText(codeElement)	コード要素からテキストを抽出
writeToClipboard(text)	クリップボードにテキストをコピー
handleGood()	いいね投票 (楽観的更新)

依存関係:

- react : useLocation , useParams , useEffect , useMemo , useRef , useState , useCallback , startTransition
- react-router-dom : Link
- mermaid
- ../data/posts.json
- ../components/AccessCounter , ../components/PrefetchLink

src/routes/Products.tsx

プロダクト一覧ページ。

型定義:

```
type Product = {
  slug: string
  name: string
  description: string
  language: string
  tags?: string[]
  url: string
  demo?: string
}
```

定数:

```
const languageColors: Record<string, string> = {
  TypeScript: '#3178c6',
  JavaScript: '#f7df1e',
  Python: '#3572A5',
  Java: '#b07219',
  Go: '#00ADD8',
  Rust: '#dea584',
}
```

機能:

- CMS APIからプロダクト一覧を取得
- 言語カラーインジケーター
- 管理者UI

依存関係:

- react : useEffect , useRef , useState
- react-router-dom : Link
- ../data/products.json
- ../components/AccessCounter , ../components/PrefetchLink
- ../contexts/AdminAuthContext

src/routes/ProductDetail.tsx

プロダクト詳細ページ。

機能:

- Markdown記事がある場合は表示、なければJSON contentを表示
- GitHub/Demoリンク
- Mermaidダイアグラム対応
- コードコピーボタン
- スクロール時背景ブラー

依存関係:

- react : useEffect , useRef
- react-router-dom : useParams , Link
- mermaid
- ../data/products.json , ../data/product-posts.json
- ../components/AccessCounter , ../components/PrefetchLink

src/routes/Photos.tsx

写真ギャラリーページ。

内部コンポーネント:

コンポーネント名	説明
PhotoCard	写真カード（クリックでLightbox表示）
PhotoDetails	Lightbox内の写真詳細情報
Tag	タグチップ

定数:

```
const RATIO_CLASS_MAP: Record<PhotoRatio, string> = {
  portrait: 'aspect-[4/5]',
  landscape: 'aspect-[4/3]',
  square: 'aspect-square',
}
```

依存関係:

- react : useState , useEffect , useRef
- react-router-dom : Link
- ../components/AccessCounter , ../components/PrefetchLink , ../components/Lightbox
- ../data/photos

src/routes/BBSList.tsx

BBS スレッド一覧ページ。

型定義:

```
type Thread = {
  id: string
  title: string
  createdAt: string
  createdBy: string
  postCount: number
  lastPostAt: string
}
```

機能:

- スレッド一覧表示

- スレッド作成フォーム
- 管理者ログイン/ログアウト
- 管理者によるスレッド削除

関数:

関数名	説明
fetchThreads()	スレッド一覧を取得
handleCreateThread(e)	新規スレッド作成
formatDisplayDate(isoDate)	日時をフォーマット
handleLogin()	管理者ログイン
handleDeleteThread(threadId, e)	スレッド削除

依存関係:

- react : useEffect , useRef , useState , useCallback
- react-router-dom : Link
- ../components/AccessCounter , ../components/PrefetchLink
- ../contexts/AdminAuthContext

src/routes/BBSThread.tsx

BBS スレッド詳細ページ。

型定義:

```
type Thread = {
  id: string
  title: string
  createdAt: string
  createdBy: string
  postCount: number
  lastPostAt: string
}

type Post = {
  id: number
  name: string
  date: string
  userId: string
  content: string
}
```

関数:

関数名	説明
parseContent(content)	アンカーリンク (>>数字) をパース
fetchThread()	スレッド・投稿を取得
handleSubmit(e)	投稿送信
handleDeletePost(postId)	投稿削除 (管理者用)

機能:

- 投稿一覧表示
- アンカーリンク (>>数字) のクリックでスムーズスクロール
- 投稿フォーム
- 管理者による投稿削除

依存関係:

- react : useEffect , useRef , useState , useCallback
- react-router-dom : useParams , Link
- ../components/AccessCounter , ../components/PrefetchLink
- ../contexts/AdminAuthContext

管理者機能

src/routes/admin/PostEditor.tsx

記事編集ページ。

型定義:

```
type PostData = {
  slug: string
  title: string
  summary: string
  markdown: string
  html: string
  tags: string[]
  createdAt: string
  updatedAt: string
}
```

関数:

関数名	説明
markdownToHtml(markdown)	MarkdownをHTMLに変換
saveCurrentDraft()	現在のデータを下書き保存
handleSave()	記事をCMS APIに保存
discardDraft()	下書きを破棄してリロード
handleDelete()	記事を削除

機能:

- 新規作成/編集の両対応
- 下書きの自動保存・復元
- ログアウト前に下書き自動保存
- Markdownエディタ
- フォーム検証

依存関係:

- react : useEffect , useState , useCallback , useRef
- react-router-dom : useParams , useNavigate , Link
- unified , remark-parse , remark-gfm , remark-rehype , rehype-stringify
- ../../contexts/AdminAuthContext
- ../../components/admin/MarkdownEditor
- ../../lib/draftStorage

src/routes/admin/ProductEditor.tsx

プロダクト編集ページ。

型定義:

```
type ProductData = {
  slug: string
  name: string
  description: string
  language: string
  tags: string[]
  url: string
  demo?: string
  markdown?: string
  html?: string
  createdAt: string
  updatedAt: string
}
```

機能:

- PostEditorと同様の機能
- プロダクト固有のフィールド（言語、URL、Demo URL）

依存関係:

- PostEditorと同様

ライブラリ・ユーティリティ

src/lib/firebase.ts

Firebase初期化。

```
const firebaseConfig = {
  apiKey: import.meta.env.VITE_FIREBASE_API_KEY,
  authDomain: import.meta.env.VITE_FIREBASE_AUTH_DOMAIN,
  projectId: import.meta.env.VITE_FIREBASE_PROJECT_ID,
  storageBucket: import.meta.env.VITE_FIREBASE_STORAGE_BUCKET,
  messagingSenderId: import.meta.env.VITE_FIREBASE_MESSAGING_SENDER_ID,
  appId: import.meta.env.VITE_FIREBASE_APP_ID,
}

export const auth = getAuth(app)
export const googleProvider = new GoogleAuthProvider()
```

依存関係:

- firebase/app : initializeApp
- firebase/auth : getAuth, GoogleAuthProvider

src/lib/draftStorage.ts

下書き保存ユーティリティ。localStorageを使用。

型定義:

```
export type PostDraft = {
  slug: string
  title: string
  summary: string
  tags: string
  markdown: string
  savedAt: number
}

export type ProductDraft = {
  slug: string
  name: string
  description: string
  language: string
  tags: string
  url: string
  demo: string
  markdown: string
  savedAt: number
}
```

関数:

関数名	引数	戻り値	説明
getDraftKey(type, slug)	'post' 'product' , string	string	下書きキーを生成
saveDraft<T>(type, slug, data)	...	void	下書きを保存
loadDraft<T>(type, slug)	...	T null	下書きを読み込み
deleteDraft(type, slug)	...	void	下書きを削除
getAllDrafts()	-	{ posts, products }	全下書きを取得
hasDraft(type, slug)	...	boolean	下書きが存在するか
formatDraftDate(savedAt)	number	string	保存日時をフォーマット

定数:

```
const DRAFT_PREFIX = 'haroin57_draft_'
```

データファイル

src/data/photos.ts

写真データ定義。

型定義:

```
export type PhotoRatio = 'portrait' | 'landscape' | 'square'

export type Photo = {
  src: string      // 画像ファイルパス
  title: string    // 写真タイトル
  location: string // 撮影場所
  date: string     // 撮影日 (YYYY-MM-DD形式)
  camera: string   // カメラ種類
  lens: string     // レンズ情報
  exposure: string // 露出情報
  note: string     // 写真の説明文
  ratio: PhotoRatio // アスペクト比
  tone: string     // アクセントカラー (HEX形式)
}
```

エクスポート:

- photos: Photo[] - 写真データ配列
- shotTags: string[] - タグ配列

初学者向け：TypeScript/JavaScript基礎文法

本プロジェクトのコードを理解するために必要な基礎文法を解説します。

変数宣言

const - 再代入不可の変数

```
const name = 'haroin57'          // 文字列
const count = 42                 // 数値
const isActive = true            // 真偽値
const items = [1, 2, 3]           // 配列
const user = { id: 1 }            // オブジェクト

// ✗ 再代入はエラー
// name = 'other' // Error: Assignment to constant variable

// ✓ 配列やオブジェクトの中身は変更可能
items.push(4)                   // [1, 2, 3, 4]
user.id = 2                      // { id: 2 }
```

let - 再代入可能な変数

```
let count = 0
count = count + 1    // ✓ 再代入OK
count++             // ✓ インクリメントOK
```

使い分け: 基本は const を使い、再代入が必要な場合のみ let を使用。

アロー関数 (Arrow Function)

従来の function 宣言の代わりに、`=>` を使った簡潔な関数定義。

基本構文

```
// 従来の関数
function add(a, b) {
  return a + b
}

// アロー関数（同じ意味）
const add = (a, b) => {
  return a + b
}

// 1行で書ける場合はさらに省略可能（暗黙のreturn）
const add = (a, b) => a + b

// 引数が1つなら括弧も省略可能
const double = x => x * 2

// 引数がないなら空括弧
const greet = () => 'Hello!'
```

本プロジェクトでの例

```
// BBSList.tsx - イベントハンドラ
const handleSubmit = (e: React.FormEvent) => {
  e.preventDefault()
  // 処理...
}

// Photos.tsx - 配列のフィルタリング
const filteredPhotos = photos.filter(photo => photo.tags.includes(selectedTag))

// AnimatedRoutes.tsx - 遅延読み込み関数
const loadHome = () => import('../routes/Home')
```

分割代入 (Destructuring)

配列やオブジェクトから値を取り出して変数に代入する構文。

配列の分割代入

```
const colors = ['red', 'green', 'blue']

// 従来の書き方
const first = colors[0] // 'red'
const second = colors[1] // 'green'

// 分割代入（同じ意味）
const [first, second] = colors // first='red', second='green'

// useStateで使われる形
const [count, setCount] = useState(0)
// count: 現在の値
// setCount: 更新関数
```

オブジェクトの分割代入

```
const user = { name: 'haroin57', age: 25, email: 'test@example.com' }

// 従来の書き方
const name = user.name
const age = user.age

// 分割代入（同じ意味）
const { name, age } = user // name='haroin57', age=25

// 関数の引数で直接分割代入
function greet({ name, age }: { name: string; age: number }) {
  return `Hello, ${name}! You are ${age} years old.`
}

// 本プロジェクトでの例 (Lightbox.tsx)
function Lightbox({ isOpen, onClose, imageSrc, imageAlt }: LightboxProps) {
  // propsから直接値を取り出している
}
```

スプレッド構文 (Spread Syntax)

... を使って配列やオブジェクトを展開する構文。

配列のスプレッド

```
const arr1 = [1, 2, 3]
const arr2 = [4, 5, 6]

// 配列を結合
const combined = [...arr1, ...arr2] // [1, 2, 3, 4, 5, 6]

// 先頭に要素を追加
const newArr = [0, ...arr1] // [0, 1, 2, 3]

// 末尾に要素を追加
const newArr2 = [...arr1, 4] // [1, 2, 3, 4]
```

オブジェクトのスプレッド

```
const user = { name: 'haroin57', age: 25 }

// オブジェクトをコピーして一部を上書き
const updatedUser = { ...user, age: 26 }
// { name: 'haroin57', age: 26 }

// 本プロジェクトでの例 (PostEditor.tsx)
setFormData((prev) => ({ ...prev, title: e.target.value }))
// prevの全プロパティをコピーし、titleだけ上書き
```

テンプレートリテラル (Template Literal)

バッククオート (`) を使った文字列。変数の埋め込みや改行が可能。

```

const name = 'haroin57'
const age = 25

// 従来の文字列結合
const message = 'Hello, ' + name + '! You are ' + age + ' years old.'

// テンプレートリテラル（同じ意味）
const message = `Hello, ${name}! You are ${age} years old.`

// 複数行も可能
const multiLine = `
  First line
  Second line
  Third line
`


// 本プロジェクトでの例 (PostDetail.tsx)
document.title = `${post.title} | haroin57`


// APIエンドポイント (BBSList.tsx)
const res = await fetch(`${BBS_ENDPOINT}/threads/${id}`)

```

三項演算子 (Ternary Operator)

条件 ? 真の場合 : 偽の場合 という形式の条件分岐。

```

const age = 20

// if文
let status
if (age >= 18) {
  status = '成人'
} else {
  status = '未成年'
}

// 三項演算子（同じ意味）
const status = age >= 18 ? '成人' : '未成年'

// 本プロジェクトでの例 (AccessCounter.tsx)
return <span>Access: {count ?? '...'}</span>
// countがnullかundefinedなら '...' を表示

// ネストした三項演算子（読みにくいので注意）
const color = score >= 90 ? 'gold' : score >= 70 ? 'silver' : 'bronze'

```

Null合体演算子（??）とオプショナルチェーン（?.）

Null合体演算子（??）

左辺が `null` または `undefined` の場合のみ右辺を返す。

```
const value = null
const result = value ?? 'default' // 'default'

const zero = 0
const result2 = zero ?? 'default' // 0 (0はnullでもundefinedでもない)

// 本プロジェクトでの例 (AccessCounter.tsx)
const cached = Number(localStorage.getItem(CACHE_KEY) ?? '0')
// localStorageがない場合は '0' を使用
```

オプショナルチェーン（?.）

プロパティが存在しない場合にエラーにならず `undefined` を返す。

```
const user = { profile: { name: 'haroin57' } }

// 従来の書き方 (エラー防止)
const name = user && user.profile && user.profile.name

// オプショナルチェーン (同じ意味)
const name = user?.profile?.name // 'haroin57'

// プロパティがない場合
const missing = user?.settings?.theme // undefined (エラーにならない)

// 本プロジェクトでの例 (Home.tsx)
const targets = pageRef.current?.querySelectorAll('.reveal')
// pageRef.currentがnullなら処理をスキップ
```

非同期処理（`async/await`）

時間のかかる処理（API呼び出し等）を待つための構文。

Promise基礎

```
// Promiseを返す関数
function fetchData() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve('データ取得完了')
    }, 1000)
  })
}

// .then()で結果を受け取る（従来の書き方）
fetchData()
  .then(result => console.log(result))
  .catch(error => console.error(error))
```

async/await（推奨）

```
// async関数内でawaitを使用
async function loadData() {
  try {
    const result = await fetchData() // 完了まで待つ
    console.log(result)
  } catch (error) {
    console.error(error)
  }
}

// 本プロジェクトでの例（BBSList.tsx）
const fetchThreads = useCallback(async () => {
  try {
    setIsLoading(true)
    const res = await fetch(`#${BBS_ENDPOINT}/threads`)
    if (!res.ok) throw new Error('Failed to fetch')
    const data = await res.json()
    setThreads(data.threads)
  } catch (err) {
    setError('取得に失敗しました')
  } finally {
    setIsLoading(false)
  }
}, [])
```

重要な注意点

```
// ✗ 間違い: awaitはasync関数の中でのみ使用可能
function wrong() {
  const data = await fetch('/api/data') // SyntaxError
}

// ✓ 正しい
async function correct() {
  const data = await fetch('/api/data')
}

// トップレベルawait (モジュールの最上位では使用可能)
const data = await fetch('/api/data') // ES2022以降
```

型注釈 (Type Annotation)

TypeScriptでは変数や関数に型を指定できる。

基本的な型

```
// プリミティブ型
const name: string = 'haroin57'
const age: number = 25
const isActive: boolean = true

// 配列
const numbers: number[] = [1, 2, 3]
const names: string[] = ['a', 'b', 'c']
const items: Array<string> = ['a', 'b'] // 同じ意味

// オブジェクト
const user: { name: string; age: number } = { name: 'haroin57', age: 25 }

// nullを許容
const value: string | null = null

// 関数の型
const greet: (name: string) => string = (name) => `Hello, ${name}!`
```

型エイリアス (type)

```
// 型に名前をつける
type User = {
  id: number
  name: string
  email: string
}

const user: User = { id: 1, name: 'haroin57', email: 'test@example.com' }

// 本プロジェクトでの例 (Lightbox.tsx)
type LightboxProps = {
  isOpen: boolean
  onClose: () => void
  imageSrc: string
  imageAlt: string
  children?: React.ReactNode // ?は省略可能を意味
}
```

interface

```
// typeと似ているが、拡張が可能
interface User {
  id: number
  name: string
}

interface AdminUser extends User {
  role: 'admin'
  permissions: string[]
}

// 本プロジェクトでの例 (PrefetchLink.tsx)
interface PrefetchLinkProps extends LinkProps {
  enablePrefetch?: boolean
}
```

ジェネリクス (Generics)

型をパラメータとして受け取る仕組み。

```
// Tは任意の型を表すプレースホルダー
function identity<T>(value: T): T {
  return value
}

identity<string>('hello') // string型
identity<number>(42) // number型
identity(true) // 型推論でboolean型

// useStateでの使用例
const [count, setCount] = useState<number>(0)
const [user, setUser] = useState<User | null>(null)
const [items, setItems] = useState<Item[]>([])
```

配列メソッド

map - 各要素を変換

```
const numbers = [1, 2, 3]
const doubled = numbers.map(n => n * 2) // [2, 4, 6]

// Reactでのリストレンダリング
const items = ['A', 'B', 'C']
return (
  <ul>
    {items.map((item, index) => (
      <li key={index}>{item}</li>
    )))
  </ul>
)
```

filter - 条件に合う要素を抽出

```
const numbers = [1, 2, 3, 4, 5]
const evens = numbers.filter(n => n % 2 === 0) // [2, 4]

// 本プロジェクトでの例 (Photos.tsx)
const filteredPhotos = photos.filter(photo =>
  photo.tags.includes(selectedTag)
)
```

find - 条件に合う最初の要素を取得

```
const users = [
  { id: 1, name: 'Alice' },
  { id: 2, name: 'Bob' }
]
const bob = users.find(user => user.name === 'Bob') // { id: 2, name: 'Bob' }

// 本プロジェクトでの例 (PostDetail.tsx)
const post = postsData.posts.find(p => p.slug === slug)
```

some / every - 条件判定

```
const numbers = [1, 2, 3, 4, 5]

// some: 1つでも条件を満たせばtrue
numbers.some(n => n > 3) // true

// every: 全てが条件を満たせばtrue
numbers.every(n => n > 0) // true
numbers.every(n => n > 3) // false
```

インポート/エクスポート (ES Modules)

名前付きエクスポート/インポート

```
// utils.ts
export function add(a: number, b: number) {
  return a + b
}
export const PI = 3.14159

// 使用側
import { add, PI } from './utils'
```

デフォルトエクスポート/インポート

```
// Component.tsx
function MyComponent() {
  return <div>Hello</div>
}
export default MyComponent

// 使用側（名前は自由）
import MyComponent from './Component'
import AnyName from './Component' // 同じものをインポート
```

本プロジェクトでの例

```
// 名前付きインポート (react)
import { useState, useEffect, useCallback } from 'react'

// デフォルトインポート (コンポーネント)
import PrefetchLink from '../components/PrefetchLink'

// 型のみのインポート
import type { LinkProps } from 'react-router-dom'

// JSON with type assertion
import postsData from '../data/posts.json' with { type: 'json' }
```

JavaScript/TypeScript標準関数・メソッド

本プロジェクトで使用されている標準APIを解説します。

文字列メソッド

`String.prototype.trim()`

文字列の前後の空白を削除する。

```
const input = ' hello world '
input.trim() // 'hello world'

// 本プロジェクトでの例 (BBSList.tsx)
if (!title.trim() || !content.trim()) return // 空白のみの入力を拒否
```

`String.prototype.split()`

文字列を指定した区切り文字で分割し、配列にする。

```
const str = 'apple,banana,cherry'
str.split(',') // ['apple', 'banana', 'cherry']

const path = '/posts/my-article'
path.split('/') // ['', 'posts', 'my-article']

// 本プロジェクトでの例 (PrefetchLink.tsx)
const pathname = raw.split('#')[0]?.split('?')[0] // ハッシュとクエリを除去
```

String.prototype.startsWith() / endsWith()

文字列が指定した文字列で始まる/終わるかを判定。

```
const url = '/posts/my-article'
url.startsWith('/posts/') // true
url.endsWith('.html') // false
```

```
// 本プロジェクトでの例 (PrefetchLink.tsx)
if (path.startsWith('/posts/')) {
  // 投稿詳細ページのプリフェッч
}
```

String.prototype.includes()

文字列に指定した文字列が含まれるかを判定。

```
const message = 'Hello, World!'
message.includes('World') // true
message.includes('world') // false (大文字小文字を区別)

// 本プロジェクトでの例 (Photos.tsx)
photo.tags.includes(selectedTag) // タグが含まれているか
```

String.prototype.slice()

文字列の一部を切り出す。

```
const str = 'Hello, World!'
str.slice(0, 5) // 'Hello' (0番目から5文字)
str.slice(7) // 'World!' (7番目から最後まで)
str.slice(-6) // 'World!' (末尾から6文字)

// 本プロジェクトでの例 (MermaidRenderer.tsx)
const id = `mermaid-${Math.random().toString(36).slice(2, 11)}`
```

String.prototype.replace() / replaceAll()

文字列の一部を置換する。

```
const text = 'Hello World'
text.replace('World', 'React')      // 'Hello React' (最初の1つだけ)
text.replaceAll('l', 'L')           // 'HeLLo WorLD' (すべて置換)

// 正規表現も使用可能
'a1b2c3'.replace(/[0-9]/g, 'X')  // 'aXbXcX'
```

配列メソッド（詳細）

Array.prototype.map()

各要素を変換した新しい配列を作成。

```
const numbers = [1, 2, 3]
numbers.map(n => n * 2) // [2, 4, 6]

// オブジェクト配列から特定のプロパティを抽出
const users = [{ name: 'Alice', age: 25 }, { name: 'Bob', age: 30 }]
users.map(u => u.name) // ['Alice', 'Bob']

// インデックスも取得可能
['a', 'b', 'c'].map((item, index) => `${index}: ${item}`)
// ['0: a', '1: b', '2: c']
```

Array.prototype.filter()

条件に合う要素だけを抽出した新しい配列を作成。

```
const numbers = [1, 2, 3, 4, 5]
numbers.filter(n => n > 2)      // [3, 4, 5]
numbers.filter(n => n % 2 === 0) // [2, 4] (偶数のみ)

// falsyな値を除去
const mixed = [0, 1, '', 'hello', null, undefined]
mixed.filter(Boolean) // [1, 'hello']
```

Array.prototype.find() / findIndex()

条件に合う最初の要素/インデックスを返す。

```
const users = [
  { id: 1, name: 'Alice' },
  { id: 2, name: 'Bob' }
]

users.find(u => u.id === 2)          // { id: 2, name: 'Bob' }
users.find(u => u.id === 99)         // undefined (見つからない)

users.findIndex(u => u.id === 2)    // 1
users.findIndex(u => u.id === 99)   // -1 (見つからない)
```

Array.prototype.reduce()

配列を1つの値に集約する。

```
const numbers = [1, 2, 3, 4, 5]

// 合計
numbers.reduce((sum, n) => sum + n, 0) // 15

// 最大値
numbers.reduce((max, n) => Math.max(max, n), -Infinity) // 5

// オブジェクトの集計
const items = [
  { category: 'A', value: 10 },
  { category: 'B', value: 20 },
  { category: 'A', value: 30 }
]
items.reduce((acc, item) => {
  acc[item.category] = (acc[item.category] || 0) + item.value
  return acc
}, {} as Record<string, number>)
// { A: 40, B: 20 }
```

Array.prototype.sort()

配列を並べ替える（元の配列を変更する）。

```
// 文字列のソート
['banana', 'apple', 'cherry'].sort() // ['apple', 'banana', 'cherry']

// 数値のソート（比較関数が必要）
[3, 1, 4, 1, 5].sort((a, b) => a - b) // [1, 1, 3, 4, 5] (昇順)
[3, 1, 4, 1, 5].sort((a, b) => b - a) // [5, 4, 3, 1, 1] (降順)

// オブジェクト配列のソート
const posts = [
  { title: 'B', date: '2024-02-01' },
  { title: 'A', date: '2024-01-01' }
]
posts.sort((a, b) => new Date(b.date).getTime() - new Date(a.date).getTime())
// 日付の降順（新しい順）
```

Array.prototype.concat() / スプレッド構文

配列を結合する。

```
const arr1 = [1, 2]
const arr2 = [3, 4]

arr1.concat(arr2) // [1, 2, 3, 4]
[...arr1, ...arr2] // [1, 2, 3, 4] (同じ結果、こちらが一般的)
```

Array.prototype.join()

配列を文字列に結合する。

```
['a', 'b', 'c'].join('-') // 'a-b-c'
['a', 'b', 'c'].join('') // 'abc'
['a', 'b', 'c'].join(',') // 'a, b, c'
```

Array.from()

イテラブルから配列を作成する。

```
// 文字列を配列に
Array.from('hello') // ['h', 'e', 'l', 'l', 'o']

// Setを配列に
Array.from(new Set([1, 2, 2, 3])) // [1, 2, 3]

// 連番の配列を作成
Array.from({ length: 5 }, (_, i) => i) // [0, 1, 2, 3, 4]

// 本プロジェクトでの例 (AdminAuthContext.tsx)
const callbacks = Array.from(beforeLogoutCallbacksRef.current)
```

オブジェクトメソッド

`Object.keys() / Object.values() / Object.entries()`

オブジェクトのキー/値/エントリを配列として取得。

```
const user = { name: 'Alice', age: 25, city: 'Tokyo' }
```

```
Object.keys(user) // ['name', 'age', 'city']
Object.values(user) // ['Alice', 25, 'Tokyo']
Object.entries(user) // [['name', 'Alice'], ['age', 25], ['city', 'Tokyo']]
```

```
// エントリを使ったループ
for (const [key, value] of Object.entries(user)) {
  console.log(`#${key}: ${value}`)
}
```

`Object.assign()`

オブジェクトをマージする（スプレッド構文の方が一般的）。

```
const defaults = { theme: 'light', lang: 'en' }
const userSettings = { theme: 'dark' }

Object.assign({}, defaults, userSettings) // { theme: 'dark', lang: 'en' }
{ ...defaults, ...userSettings } // 同じ結果
```

数値・数学関数

`Number() / parseInt() / parseFloat()`

文字列を数値に変換する。

```
Number('42') // 42
Number('3.14') // 3.14
Number('hello') // NaN (変換失敗)
Number('') // 0

parseInt('42px') // 42 (数値部分のみ解析)
parseInt('3.14') // 3 (整数のみ)
parseFloat('3.14') // 3.14

// 本プロジェクトでの例 (AccessCounter.tsx)
const cached = Number(localStorage.getItem(CACHE_KEY) ?? '0')
```

```
Number.isFinite() / Number.isNaN()
```

数値の検証。

```
Number.isFinite(42)          // true
Number.isFinite(Infinity)    // false
Number.isFinite(NaN)         // false

Number.isNaN(NaN)           // true
Number.isNaN('hello')       // false (グローバルisNaNとは異なる)

// 本プロジェクトでの例 (AccessCounter.tsx)
if (Number.isFinite(cached) && cached > 0) { ... }
```

Math オブジェクト

数学関数のコレクション。

```
Math.random()                // 0以上1未満のランダムな小数
Math.floor(3.7)              // 3 (切り捨て)
Math.ceil(3.2)               // 4 (切り上げ)
Math.round(3.5)              // 4 (四捨五入)
Math.max(1, 5, 3)            // 5
Math.min(1, 5, 3)            // 1
Math.abs(-5)                 // 5 (絶対値)
Math.pow(2, 3)                // 8 (2の3乗)

// ランダムなIDを生成
Math.random().toString(36).slice(2, 11) // 'k5x8m2n1p'のような文字列
```

日付・時間

Date オブジェクト

日付と時間を扱う。

```

// 現在時刻
const now = new Date()
now.getTime()          // 1703001600000 (UNIXタイムスタンプ、ミリ秒)
now.toISOString()      // '2024-12-20T00:00:00.000Z'
now.toLocaleDateString('ja-JP') // '2024/12/20'

// 特定の日付を作成
new Date('2024-12-20')
new Date(2024, 11, 20) // 月は0始まり (11 = 12月)

// 日付の計算
const future = new Date(Date.now() + 60 * 60 * 1000) // 1時間後

// 本プロジェクトでの例 (AdminAuthContext.tsx)
const expiresAt = Date.now() + SESSION_TIMEOUT_MS

setTimeout() / setInterval() / clearTimeout() / clearInterval()


```

タイマー関数。

```

// 一度だけ実行
const timerId = setTimeout(() => {
  console.log('3秒経過')
}, 3000)

// キャンセル
clearTimeout(timerId)

// 繰り返し実行
const intervalId = setInterval(() => {
  console.log('1秒ごとに実行')
}, 1000)

// 停止
clearInterval(intervalId)


```

本プロジェクトでの例 (AdminAuthContext.tsx)

```

sessionTimeoutRef.current = setTimeout(async () => {
  await logout()
}, SESSION_TIMEOUT_MS)

```

Web API

`fetch()`

HTTPリクエストを送信する。

```
// GETリクエスト
const response = await fetch('/api/posts')
const data = await response.json()

// POSTリクエスト
const response = await fetch('/api/posts', {
  method: 'POST',
  headers: { 'Content-Type': 'application/json' },
  body: JSON.stringify({ title: 'New Post', content: '...' })
})

// エラーハンドリング
if (!response.ok) {
  throw new Error(`HTTP ${response.status}`)
}

// 本プロジェクトでの例 (BBSList.tsx)
const res = await fetch(`${BBS_ENDPOINT}/threads`, {
  method: 'POST',
  headers: { 'Content-Type': 'application/json' },
  body: JSON.stringify({ title, content, author }),
})
```

localStorage / sessionStorage

ブラウザにデータを保存する。

```
// 保存
localStorage.setItem('key', 'value')
localStorage.setItem('user', JSON.stringify({ name: 'Alice' }))

// 取得
const value = localStorage.getItem('key') // 'value' または null
const user = JSON.parse(localStorage.getItem('user') || '{}')

// 削除
localStorage.removeItem('key')

// 全削除
localStorage.clear()

// 本プロジェクトでの例 (AccessCounter.tsx)
localStorage.setItem(CACHE_KEY, String(data.total))
const cached = localStorage.getItem(CACHE_KEY)
```

AbortController

非同期処理のキャンセル。

```

const controller = new AbortController()

fetch('/api/data', { signal: controller.signal })
  .then(res => res.json())
  .catch(err => {
    if (err.name === 'AbortError') {
      console.log('リクエストがキャンセルされました')
    }
  })
}

// キャンセル
controller.abort()

// 本プロジェクトでの例 (AccessCounter.tsx)
useEffect(() => {
  const controller = new AbortController()

  fetch(API_ENDPOINT, { signal: controller.signal })
    .then(/* ... */)

  return () => controller.abort() // クリーンアップ時にキャンセル
}, [])

```

console メソッド

デバッグ出力。

```

console.log('通常のログ')
console.error('エラー')
console.warn('警告')
console.table([{ a: 1 }, { a: 2 }]) // 表形式で表示
console.time('処理'); /* ... */ console.timeEnd('処理') // 時間計測
console.group('グループ'); console.log('内容'); console.groupEnd()

```

DOM操作

`document.querySelector()` / `querySelectorAll()`

CSSセレクタで要素を取得。

```
// 単一要素
const header = document.querySelector('.header')
const button = document.querySelector('#submit-btn')
const input = document.querySelector('input[type="email"]')

// 複数要素
const items = document.querySelectorAll('.list-item')
items.forEach(item => console.log(item.textContent))

// 本プロジェクトでの例 (Home.tsx)
const targets = pageRef.current?.querySelectorAll('.reveal')
```

element.classList

CSSクラスの操作。

```
const el = document.querySelector('.box')

el.classList.add('active')          // クラスを追加
el.classList.remove('active')       // クラスを削除
el.classList.toggle('active')       // あれば削除、なければ追加
el.classList.contains('active')    // 含まれているか確認

// 本プロジェクトでの例 (PostDetail.tsx / ProductDetail.tsx)
const body = document.body
body.classList.add('post-detail-page')
body.classList.remove('post-detail-page')
```

addEventListener() / removeEventListener()

イベントリスナーの登録と解除。

```
const handleClick = (e: MouseEvent) => {
  console.log('クリックされた', e.target)
}

element.addEventListener('click', handleClick)
element.removeEventListener('click', handleClick)

// オプション
window.addEventListener('scroll', handleScroll, { passive: true })
window.addEventListener('click', handleClick, { once: true }) // 1回だけ

// 本プロジェクトでの例 (ScrollTopHomeSwitch.tsx)
window.addEventListener('wheel', onWheel, { passive: true })
return () => window.removeEventListener('wheel', onWheel)
```

Promise関連

`Promise.all() / Promise.race() / Promise.allSettled()`

複数のPromiseを扱う。

```
// すべて完了を待つ (1つでも失敗すると全体が失敗)
const [user, posts] = await Promise.all([
  fetch('/api/user').then(r => r.json()),
  fetch('/api/posts').then(r => r.json())
])

// 最後に完了したものを取得
const fastest = await Promise.race([
  fetch('/api/server1'),
  fetch('/api/server2')
])

// すべての結果を取得 (成功/失敗問わず)
const results = await Promise.allSettled([
  fetch('/api/a'),
  fetch('/api/b')
])
// [{ status: 'fulfilled', value: ... }, { status: 'rejected', reason: ... }]
```

JSON

`JSON.stringify() / JSON.parse()`

オブジェクトとJSON文字列の相互変換。

```
const obj = { name: 'Alice', age: 25 }

// オブジェクト → JSON文字列
JSON.stringify(obj)           // '{"name":"Alice","age":25}'
JSON.stringify(obj, null, 2)   // 整形 (インデント2)

// JSON文字列 → オブジェクト
JSON.parse('{"name":"Alice"}') // { name: 'Alice' }

// 本プロジェクトでの例 (BBSList.tsx)
body: JSON.stringify({ title, content, author })
const data = await res.json() // fetchのjson()は内部でJSON.parseを呼ぶ
```

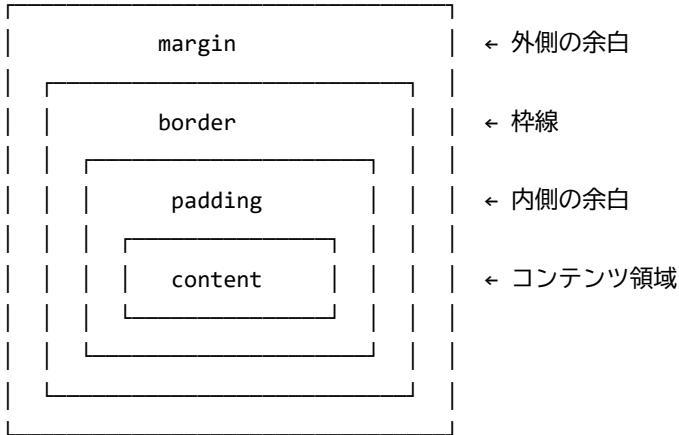
CSS基礎とTailwind CSS

本プロジェクトではTailwind CSSを使用しています。まず標準CSSの基礎を理解してからTailwindを学びましょう。

CSS基礎

ボックスモデル

すべてのHTML要素は「ボックス」として扱われる。



```
/* 標準CSS */
.box {
  width: 200px;
  height: 100px;
  padding: 16px;
  margin: 8px;
  border: 1px solid black;
}

/* Tailwind CSS */
<div className="w-[200px] h-[100px] p-4 m-2 border border-black">
```

主要CSSプロパティとTailwind対応表

レイアウト

CSS	Tailwind	説明
display: flex	flex	フレックスボックス
display: grid	grid	グリッドレイアウト

CSS	Tailwind	説明
display: block	block	ブロック要素
display: inline	inline	インライン要素
display: none	hidden	非表示
position: relative	relative	相対配置
position: absolute	absolute	絶対配置
position: fixed	fixed	固定配置
position: sticky	sticky	ステイッキー配置

Flexbox

CSS	Tailwind	説明
flex-direction: row	flex-row	横並び
flex-direction: column	flex-col	縦並び
justify-content: center	justify-center	主軸の中央揃え
justify-content: space-between	justify-between	両端揃え
align-items: center	items-center	交差軸の中央揃え
flex-wrap: wrap	flex-wrap	折り返し
gap: 16px	gap-4	要素間の隙間

```
// 本プロジェクトでの例 (Lightbox.tsx)
<div className="fixed inset-0 z-50 flex items-center justify-center">
// → position: fixed; top/right/bottom/left: 0; z-index: 50;
//   display: flex; align-items: center; justify-content: center;
```

サイズ

CSS	Tailwind	説明
width: 100%	w-full	幅100%
width: 100vw	w-screen	ビューポート幅
width: auto	w-auto	自動
max-width: 1024px	max-w-4xl	最大幅
height: 100%	h-full	高さ100%
height: 100vh	h-screen	ビューポート高さ

CSS	Tailwind	説明
min-height: 100vh	min-h-screen	最小高さ

余白 (Spacing)

Tailwindの余白は4pxを1単位とする。

CSS	Tailwind	値
padding: 0	p-0	0px
padding: 4px	p-1	4px
padding: 8px	p-2	8px
padding: 16px	p-4	16px
padding: 24px	p-6	24px
padding: 32px	p-8	32px
margin: 16px	m-4	16px
margin: auto	m-auto	自動

方向指定:

- p-4 → 全方向
- px-4 → 左右 (x軸)
- py-4 → 上下 (y軸)
- pt-4 → 上 (top)
- pr-4 → 右 (right)
- pb-4 → 下 (bottom)
- pl-4 → 左 (left)

色

```
// 背景色
<div className="bg-white">          // 白
<div className="bg-black">           // 黒
<div className="bg-gray-500">         // グレー
<div className="bg-blue-500">          // 青
<div className="bg-red-500">          // 赤
<div className="bg-transparent">        // 透明

// 透明度付き
<div className="bg-black/50">          // 黒、透明度50%
<div className="bg-white/10">           // 白、透明度10%

// 文字色
<div className="text-white">
<div className="text-gray-400">

// 本プロジェクトでの例
<div className="bg-black/90 backdrop-blur-sm">
// → background: rgba(0, 0, 0, 0.9); backdrop-filter: blur(4px);
```

テキスト

CSS	Tailwind	説明
font-size: 14px	text-sm	小さい文字
font-size: 16px	text-base	基本サイズ
font-size: 18px	text-lg	大きい文字
font-size: 24px	text-2xl	見出し
font-weight: bold	font-bold	太字
font-weight: 600	font-semibold	やや太字
text-align: center	text-center	中央揃え
text-align: left	text-left	左揃え
line-height: 1.5	leading-relaxed	行間

角丸 (Border Radius)

CSS	Tailwind	値
border-radius: 0	rounded-none	なし
border-radius: 4px	rounded	小

CSS	Tailwind	値
border-radius: 8px	rounded-lg	中
border-radius: 16px	rounded-2xl	大
border-radius: 9999px	rounded-full	円形

ボーダー

```
<div className="border">          // 1px solid
<div className="border-2">        // 2px
<div className="border-gray-300">   // 色指定
<div className="border-t">         // 上だけ
<div className="border border-white/50"> // 透明度付き
```

影 (Box Shadow)

CSS	Tailwind	説明
box-shadow: ...	shadow-sm	小さい影
box-shadow: ...	shadow	標準の影
box-shadow: ...	shadow-lg	大きい影
box-shadow: ...	shadow-xl	より大きい影
box-shadow: none	shadow-none	影なし

トランジション・アニメーション

```
// トランジション
<div className="transition">          // すべてのプロパティ
<div className="transition-colors">      // 色のみ
<div className="transition-opacity">     // 透明度のみ
<div className="duration-300">           // 300ms
<div className="ease-in-out">             // イージング

// 本プロジェクトでの例 (Lightbox.tsx)
<button className="transition-colors hover:bg-white/20">
// → マウスオーバー時に背景色がアニメーション
```

ホバー・フォーカス・状態

```
// ホバー（マウスオーバー）
<button className="hover:bg-blue-600">

// フォーカス（選択時）
<input className="focus:ring-2 focus:border-blue-500">

// アクティブ（クリック中）
<button className="active:scale-95">

// 無効状態
<button className="disabled:opacity-50" disabled>

// グループホバー
<div className="group">
  <span className="group-hover:text-blue-500">
</div>
```

レスポンシブ

画面サイズに応じてスタイルを変更。

プレフィックス	最小幅	説明
(なし)	0px	モバイルファースト
sm:	640px	小さいタブレット
md:	768px	タブレット
lg:	1024px	デスクトップ
xl:	1280px	大きいデスクトップ
2xl:	1536px	超大画面

```
// モバイルでは1列、タブレット以上では2列、デスクトップでは3列
<div className="grid grid-cols-1 md:grid-cols-2 lg:grid-cols-3">

// モバイルでは非表示、デスクトップでは表示
<div className="hidden lg:block">

// 本プロジェクトでの例 (Lightbox.tsx)
<div className="p-4 sm:p-6">
// → モバイル: padding: 16px; タブレット以上: padding: 24px;
```

よく使うTailwindパターン

センタリング

```
// Flexboxで中央揃え
<div className="flex items-center justify-center">

// 水平方向のみ中央
<div className="mx-auto max-w-4xl">

// 絶対配置で中央
<div className="absolute top-1/2 left-1/2 -translate-x-1/2 -translate-y-1/2">
```

フルスクリーンオーバーレイ

```
// 本プロジェクトでの例 (Lightbox.tsx)
<div className="fixed inset-0 z-50 flex items-center justify-center bg-black/90">
// → 画面全体を覆う半透明の黒背景
```

カード

```
<div className="rounded-lg bg-white p-4 shadow-lg">
  <h2 className="text-lg font-bold">タイトル</h2>
  <p className="text-gray-600">内容</p>
</div>
```

ボタン

```
<button className="rounded-lg bg-blue-500 px-4 py-2 text-white transition-colors hover:bg-blue-600">
  クリック
</button>
```

入力フィールド

```
<input
  className="w-full rounded-lg border border-gray-300 px-4 py-2 focus:border-blue-500 focus:outline-none focus:ring-2 focus:ring-blue-500"
  type="text"
/>
```

React基礎概念

Reactを理解するための核となる概念を解説します。

コンポーネント (Component)

UIを構成する再利用可能なパート。関数として定義する。

```
// 最もシンプルなコンポーネント
function HelloWorld() {
  return <div>Hello, World!</div>
}

// propsを受け取るコンポーネント
type GreetingProps = {
  name: string
  age?: number // オプショナル
}

function Greeting({ name, age }: GreetingProps) {
  return (
    <div>
      <p>Hello, {name}!</p>
      {age && <p>You are {age} years old.</p>}
    </div>
  )
}

// 使用例
<Greeting name="haroin57" age={25} />
```

JSX

JavaScriptの中にHTMLライクな構文を書ける拡張構文。

```

// JSXの基本
const element = <h1>Hello, World!</h1>

// JavaScript式の埋め込み ({}で囲む)
const name = 'haroin57'
const element = <h1>Hello, {name}!</h1>

// 条件付きレンダリング
function Status({ isLoggedIn }: { isLoggedIn: boolean }) {
  return (
    <div>
      {isLoggedIn ? (
        <p>ログイン中</p>
      ) : (
        <p>ログインしてください</p>
      )}
    </div>
  )
}

// 条件付き表示 (&&演算子)
function MaybeShow({ show, message }: { show: boolean; message: string }) {
  return (
    <div>
      {show && <p>{message}</p>} /* showがtrueの時のみ表示 */
    </div>
  )
}

```

フック (Hooks)

関数コンポーネントに状態や副作用などの機能を追加する関数。

```

// useで始まる関数がフック
import { useState, useEffect, useCallback } from 'react'

function Counter() {
  // フックはコンポーネントの最上位で呼び出す
  const [count, setCount] = useState(0) // ✓ OK

  // ✗ 条件分岐の中で呼び出すのはNG
  // if (someCondition) {
  //   const [value, setValue] = useState(0) // Error
  // }

  return <button onClick={() => setCount(count + 1)}>{count}</button>
}

```

再レンダリング (Re-rendering)

Reactは状態が変更されると、そのコンポーネントを再度実行（再レンダリング）する。

```
function Counter() {
  console.log('レンダリング実行') // 状態が変わるたびに出力

  const [count, setCount] = useState(0)

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={() => setCount(count + 1)}>
        +1
      </button>
    </div>
  )
}

// クリックするたびに:
// 1. setCountが呼ばれる
// 2. countの値が更新される
// 3. Counterコンポーネントが再レンダリングされる
// 4. 新しいcountの値でUIが更新される
```

依存関係一覧

外部ライブラリ詳細解説

React関連

react

Reactは、Facebookが開発したUIライブラリ。コンポーネントベースの宣言的UIを構築できる。

インポートされるAPI一覧:

API	説明	使用ファイル例
useState	コンポーネント内で状態を管理するフック	全ページコンポーネント
useEffect	副作用（API呼び出し、DOM操作等）を実行するフック	全ページコンポーネント
useCallback	コールバック関数をメモ化し、不要な再レンダリングを防ぐ	イベントハンドラ定義

API	説明	使用ファイル例
useMemo	計算結果をメモ化し、パフォーマンスを最適化	Posts.tsx , PostDetail.tsx , MarkdownEditor.tsx
useRef	ミュータブルな参照を保持。 DOM参照やタイマーID等に使用	全ページコンポーネント
useContext	Contextから値を取得するフック	AdminAuthContext.tsx
createContext	Context (コンポーネントツリー全体で値を共有する仕組み) を作成	AdminAuthContext.tsx
lazy	コンポーネントの遅延読み込み (コード分割)	AnimatedRoutes.tsx
Suspense	遅延読み込み中のフォールバックUIを表示	AnimatedRoutes.tsx
createRef	ref オブジェクトを作成	(現状未使用)
useLayoutEffect	DOM変更後、ブラウザ描画前に同期的に実行	(現状未使用)
startTransition	低優先度の状態更新をマーク (Concurrent Mode)	Posts.tsx , PostDetail.tsx
ReactNode	Reactがレンダリングできるすべての型	型定義

useState - 状態管理フック

コンポーネント内でリアクティブな状態を管理するための基本フック。状態が変更されるとコンポーネントが再レンダリングされる。

💡 初学者向け解説:

「状態 (state)」とは、コンポーネントが記憶しておきたいデータのこと。例えば：

- フォームに入力された文字
- ボタンがクリックされた回数
- APIから取得したデータ
- ローディング中かどうか

普通の変数 (`let count = 0`) は、コンポーネントが再レンダリングされるたびにリセットされてしまう。`useState` を使うと、再レンダリングしても値が保持される。

```
// ❌ 普通の変数: クリックしてもcountは常に0のまま
function Counter() {
  let count = 0 // 再レンダリングのたびに0にリセット
  return <button onClick={() => count++}>{count}</button>
}

// ✅ useState: クリックするとcountが増える
function Counter() {
  const [count, setCount] = useState(0) // 値が保持される
  return <button onClick={() => setCount(count + 1)}>{count}</button>
}
```

シグネチャ:

```
const [state, setState] = useState<T>(initialValue: T | ((() => T)))
```

構文の読み方:

- const [state, setState] → 配列の分割代入。useState は[現在の値, 更新関数]の配列を返す
- useState<T> → ジェネリクス。T は状態の型（省略すると自動推論）
- initialValue → 最初に設定される値

引数:

- initialValue : 状態の初期値。関数を渡すと遅延初期化（初回レンダリング時のみ実行）

戻り値:

- state : 現在の状態値
- setState : 状態を更新する関数。新しい値または更新関数を受け取る

本プロジェクトでの使用例:

```
// BBSList.tsx - シンプルな状態管理
const [threads, setThreads] = useState<Thread[]>([])
const [isLoading, setIsLoading] = useState(true)
const [error, setError] = useState<string | null>(null)

// フォーム入力の状態管理
const [title, setTitle] = useState('')
const [content, setContent] = useState('')

// 状態の更新
setThreads(data.threads) // 直接値を設定
setThreads((prev) => [newThread, ...prev]) // 関数形式で前の状態を参照
```

```

// PostEditor.tsx - オブジェクト状態の管理
const [formData, setFormData] = useState({
  slug: '',
  title: '',
  summary: '',
  tags: '',
})

// オブジェクトの一部を更新（スプレッド構文を使用）
setFormData((prev) => ({ ...prev, title: e.target.value }))

// AccessCounter.tsx - 遅延初期化
const cached = typeof window !== 'undefined'
  ? Number(localStorage.getItem(CACHE_KEY) ?? '0')
  : 0
const [count, setCount] = useState<number | null>(
  Number.isFinite(cached) && cached > 0 ? cached : null
)

```

注意点:

- 状態更新は非同期的にバッチ処理される
- オブジェクトや配列を更新する際は、新しい参照を作成する必要がある（イミュータブル更新）
- 前の状態に依存する更新は関数形式を使用する

useEffect - 副作用フック

コンポーネントのレンダリング後に副作用（API呼び出し、DOM操作、イベントリスナー登録等）を実行するフック。

💡 初学者向け解説:

「副作用（side effect）」とは、コンポーネントの描画（レンダリング）以外の処理のこと：

- サーバーからデータを取得する（fetch）
- ページタイトルを変更する（document.title）
- タイマーを設定する（setTimeout）
- イベントリスナーを登録する（addEventListener）
- ローカルストレージに保存する

これらは「画面を描く」という本来の役割とは別の「副次的な効果」なので副作用と呼ぶ。

```
// 基本形: コンポーネントが表示された時に何かする
useEffect(() => {
  console.log('コンポーネントが表示されました！')
}, [])

// クリーンアップ付き: コンポーネントが消える時に後片付け
useEffect(() => {
  const timer = setInterval(() => console.log('tick'), 1000)

  // この関数はコンポーネントが消える時に呼ばれる
  return () => {
    clearInterval(timer) // タイマーを停止
    console.log('クリーンアップ完了')
  }
}, [])


```

シグネチャ:

```
useEffect(effect: () => void | ((() => void), deps?: DependencyList)
```

構文の読み方:

- effect → 実行したい処理を書いた関数（アロー関数で書くことが多い）
- () => void | ((() => void) → 戻り値は「なし」または「クリーンアップ関数」
- deps → 依存配列。省略可能だが通常は指定する

引数:

- effect : 副作用を実行する関数。クリーンアップ関数を返すことができる
- deps : 依存配列。この配列内の値が変更された時のエフェクトが再実行される

依存配列のパターン:

書き方	意味	実行タイミング
[]	空配列	マウント時に1回だけ
[value]	値を指定	マウント時 + valueが変わった時
[a, b]	複数指定	マウント時 + aまたはbが変わった時
省略	なし	毎回のレンダリング後（非推奨）

```

// パターン1: マウント時のみ (APIから初期データ取得など)
useEffect(() => {
  fetchData()
}, []) // 空配列 = 一度だけ

// パターン2: 依存値が変わった時 (検索条件が変わったら再検索など)
useEffect(() => {
  search(query)
}, [query]) // queryが変わるたびに実行

// パターン3: 複数の依存値
useEffect(() => {
  updateFilter(category, sort)
}, [category, sort]) // どちらかが変わったら実行

```

本プロジェクトでの使用例:

```

// AdminAuthContext.tsx - Firebase認証状態の監視
useEffect(() => {
  const unsubscribe = onAuthStateChanged(auth, async (firebaseUser) => {
    setUser(firebaseUser)
    if (firebaseUser) {
      const token = await firebaseUser.getIdToken()
      setIdToken(token)
    } else {
      setIdToken(null)
    }
    setIsLoading(false)
  })
  // クリーンアップ: コンポーネントアンマウント時に購読解除
  return () => unsubscribe()
}, []) // 空配列: マウント時に1回だけ実行

// BBSList.tsx - データフェッチ
useEffect(() => {
  fetchThreads()
}, [fetchThreads]) // fetchThreadsが変更された時に再実行

```

```

// ScrollTopHomeSwitch.tsx - イベントリスナーの登録と解除
useEffect(() => {
  const onWheel = (event: WheelEvent) => { /* ... */ }
  const onTouchStart = (event: TouchEvent) => { /* ... */ }

  window.addEventListener('wheel', onWheel, { passive: true })
  window.addEventListener('touchstart', onTouchStart, { passive: true })

  // クリーンアップ: イベントリスナーの解除
  return () => {
    window.removeEventListener('wheel', onWheel)
    window.removeEventListener('touchstart', onTouchStart)
  }
}, [location.pathname, navigate])

// PostDetail.tsx - ページタイトルの設定
useEffect(() => {
  if (post) {
    document.title = `${post.title} | haroin57`
  }
}, [post])

// AdminAuthContext.tsx - タイマーのクリーンアップ
useEffect(() => {
  return () => {
    if (sessionTimeoutRef.current) {
      clearTimeout(sessionTimeoutRef.current)
    }
  }
}, [])

```

注意点:

- クリーンアップ関数は、依存配列の値が変更される前と、コンポーネントアンマウント時に実行される
- 依存配列に含めるべき値を省略すると、古い値を参照するバグ（stale closure）が発生する
- ESLintの `react-hooks/exhaustive-deps` ルールで依存配列を検証できる

useCallback - コールバック関数メモ化フック

関数をメモ化し、依存配列が変更されない限り同じ関数参照を返す。子コンポーネントへの不要な再レンダリングを防止。

💡 初学者向け解説:

「メモ化（memoization）」とは、計算結果を記憶しておいて再利用する最適化技法のこと。

なぜ必要？

Reactでは、コンポーネントが再レンダリングされるたびに、その内で定義された関数も新しく作り直される。

```
function Parent() {
  // 再レンダリングのたびに新しい関数オブジェクトが作られる
  const handleClick = () => { console.log('clicked') }

  // Childに渡されるhandleClickは毎回「別の関数」と見なされる
  return <Child onClick={handleClick} />
}
```

これが問題になるのは：

1. 子コンポーネントが「propsが変わった」と判断して不要な再レンダリングをする
2. `useEffect` の依存配列に関数を入れると、毎回エフェクトが実行される

```
function Parent() {
  // useCallbackで関数をメモ化
  // 依存配列が変わらない限り、同じ関数参照を返す
  const handleClick = useCallback(() => {
    console.log('clicked')
  }, []) // 依存なし = ずっと同じ関数

  return <Child onClick={handleClick} /> // 同じ参照なので Child は再レンダリングされない
}
```

シグネチャ:

```
const memoizedCallback = useCallback(callback: T, deps: DependencyList): T
```

構文の読み方:

- `callback` → メモ化したい関数
- `deps` → 依存配列 (`useEffect`と同じ考え方)
- 戻り値 → メモ化された関数 (同じ参照)

本プロジェクトでの使用例:

```
// App.tsx - ナビゲーション関数のメモ化
const handleNavigate = useCallback(() => {
  navigate('/home')
}, [navigate]) // navigateが変更された時のみ新しい関数を作成
```

```

// BBSList.tsx - APIコール関数のメモ化
const fetchThreads = useCallback(async () => {
  try {
    setIsLoading(true)
    const res = await fetch(`/${BBS_ENDPOINT}/threads`)
    if (!res.ok) throw new Error('Failed to fetch threads')
    const data = await res.json() as { threads: Thread[] }
    setThreads(data.threads || [])
    setError(null)
  } catch (err) {
    console.error('Failed to fetch threads:', err)
    setError('スレッド一覧の取得に失敗しました')
  } finally {
    setIsLoading(false)
  }
}, []) // 依存なし: 常に同じ関数

// フォーム送信ハンドラのメモ化
const handleCreateThread = useCallback(
  async (e: React.FormEvent) => {
    e.preventDefault()
    if (!title.trim() || !content.trim() || isSubmitting) return
    // ...処理
  },
  [title, content, isSubmitting] // これらが変更された時のみ新しい関数
)

// AdminAuthContext.tsx - ログアウト関数のメモ化
const logout = useCallback(async (skipCallbacks = false) => {
  try {
    if (!skipCallbacks) {
      const callbacks = Array.from(beforeLogoutCallbacksRef.current)
      for (const callback of callbacks) {
        try {
          await callback()
        } catch (err) {
          console.error('Before logout callback failed:', err)
        }
      }
    }
    // ...ログアウト処理
  } catch (error) {
    console.error('Logout failed:', error)
  }
}, []) // refはレンダリング間で同じなので依存に含めない

```

useCallbackを使うべき場面:

1. 関数を子コンポーネントにpropsとして渡す時
2. 関数をuseEffectの依存配列に含める時
3. 高コストな計算を含む関数

注意点:

- すべての関数に使用する必要はない（過度な最適化は複雑さを増す）
- 依存配列が頻繁に変更される場合、メモ化の効果が薄れる

useMemo - 計算結果メモ化フック

計算コストの高い値をメモ化し、依存配列が変更されない限り再計算をスキップ。

💡 初学者向け解説:

`useCallback` が「関数を記憶」するのに対し、`useMemo` は「関数の実行結果を記憶」する。

なぜ必要？

重い計算処理があると、再レンダリングのたびに毎回計算が走ってしまう。

```
function ProductList({ products, filter }) {
  // ❌ 毎回のレンダリングでフィルタリングが実行される
  const filteredProducts = products.filter(p => p.category === filter)

  return <ul>{filteredProducts.map(p => <li key={p.id}>{p.name}</li>)}</ul>
}
```

`useMemo` を使うと、依存値が変わった時だけ計算が実行される。

```
function ProductList({ products, filter }) {
  // ✅ productsかfilterが変わった時だけ計算
  const filteredProducts = useMemo(() => {
    return products.filter(p => p.category === filter)
  }, [products, filter])

  return <ul>{filteredProducts.map(p => <li key={p.id}>{p.name}</li>)}</ul>
}
```

useCallback との違い:

```
// useMemo: 計算結果を記憶
const doubled = useMemo(() => numbers.map(n => n * 2), [numbers])
// → doubled は配列 [2, 4, 6, ...]

// useCallback: 関数を記憶
const double = useCallback((n) => n * 2, [])
// → double は関数 (n) => n * 2
```

シグネチャ:

```
const memoizedValue = useMemo<T>(factory: () => T, deps: DependencyList): T
```

構文の読み方:

- factory → 値を計算する関数（この関数の「戻り値」がメモ化される）
- deps → 依存配列
- 戻り値 → factory関数の実行結果

本プロジェクトでの使用例:

```
// Posts.tsx - フィルタリングされた記事リストのメモ化
const filteredPosts = useMemo(() => {
  if (!selectedTag) return posts
  return posts.filter(post => post.tags?.includes(selectedTag))
}, [posts, selectedTag]) // postsまたはselectedTagが変更された時のみ再計算

// MarkdownEditor.tsx - Frontmatterパース結果のメモ化
const { data, content } = useMemo(
  () => parseFrontmatter(source),
  [source] // sourceが変更された時のみ再パース
)
```

useCallbackとuseMemoの違い:

```
// useCallback: 関数自体をメモ化
const fn = useCallback(() => doSomething(a, b), [a, b])

// useMemo: 関数の実行結果をメモ化
const value = useMemo(() => computeExpensiveValue(a, b), [a, b])

// useCallback(fn, deps) は useMemo(() => fn, deps) と等価
```

useRef - ミュータブル参照フック

レンダリング間で保持されるミュータブルな参照を作成。値を変更しても再レンダリングをトリガーしない。

💡 初学者向け解説:

`useRef` は「箱」のようなもの。中に何でも入れられ、中身を変えても React は気にしない（再レンダリングしない）。

useStateとの違い:

特性	<code>useState</code>	<code>useRef</code>
値の変更	再レンダリングする	再レンダリングしない
値の読み方	<code>value</code>	<code>ref.current</code>
主な用途	UIに表示する値	DOM参照、内部フラグ、タイマーID

主な使い方3パターン:

```

// パターン1: DOM要素への参照
function InputFocus() {
  const inputRef = useRef<HTMLInputElement>(null)

  const handleClick = () => {
    inputRef.current?.focus() // inputにフォーカスを当てる
  }

  return (
    <>
      <input ref={inputRef} />
      <button onClick={handleClick}>フォーカス</button>
    </>
  )
}

// パターン2: 値を保持（再レンダリングなしで）
function Timer() {
  const countRef = useRef(0) // 内部カウンター

  useEffect(() => {
    const id = setInterval(() => {
      countRef.current++ // 更新してもUIは変わらない
      console.log(countRef.current)
    }, 1000)
    return () => clearInterval(id)
  }, [])

  return <div>コンソールを確認</div>
}

// パターン3: 前回の値を保持
function Counter() {
  const [count, setCount] = useState(0)
  const prevCountRef = useRef<number>()

  useEffect(() => {
    prevCountRef.current = count // レンダリング後に更新
  })

  return (
    <div>
      現在: {count}, 前回: {prevCountRef.current}
    </div>
  )
}

```

シグネチャ:

```
const ref = useRef<T>(initialValue: T): MutableRefObject<T>
const ref = useRef<T>(null): RefObject<T> // DOM要素用
```

構文の読み方:

- initialValue → 初期値 (.current の初期値)
- .current → 値を読み書きするプロパティ

主な用途:

1. DOM要素への参照
2. タイマーID等のミュータブル値の保持
3. 前回の値の保持
4. レンダリングをトリガーしない状態の保持

本プロジェクトでの使用例:

```
// DOM要素への参照
const pageRef = useRef<HTMLDivElement | null>(null)
// JSX内で使用
<div ref={pageRef}>...</div>
// DOM操作
const targets = pageRef.current?.querySelectorAll('.reveal')

// AdminAuthContext.tsx - タイマーIDの保持
const sessionTimeoutRef = useRef<ReturnType<typeof setTimeout> | null>(null)

// タイマー設定
sessionTimeoutRef.current = setTimeout(async () => {
  await logout()
}, SESSION_TIMEOUT_MS)

// タイマークリア
if (sessionTimeoutRef.current) {
  clearTimeout(sessionTimeoutRef.current)
  sessionTimeoutRef.current = null
}

// AdminAuthContext.tsx - コールバック関数セットの保持
const beforeLogoutCallbacksRef = useRef<Set<BeforeLogoutCallback>>(new Set())

// コールバック登録
beforeLogoutCallbacksRef.current.add(callback)

// コールバック削除
beforeLogoutCallbacksRef.current.delete(callback)
```

```
// AccessCounter.tsx - 重複実行防止フラグ
const didSend = useRef(false)

useEffect(() => {
  if (didSend.current) return // 既に送信済みならスキップ
  didSend.current = true
  // APIコール...
}, [])
```

```
// ScrollTopHomeSwitch.tsx - 累積値の保持
const wheelAccumRef = useRef(0)
const wheelLastAtRef = useRef(0)

// イベントハンドラ内で更新（再レンダリングなし）
wheelAccumRef.current += deltaY
wheelLastAtRef.current = now
```

```
// AnimatedRoutes.tsx - マウント状態の追跡
const hasMountedRef = useRef(false)

useLayoutEffect(() => {
  if (!hasMountedRef.current) {
    hasMountedRef.current = true
    return // 初回はスキップ
  }
  // 2回目以降の処理...
}, [location.pathname])
```

useStateとuseRefの使い分け:

特性	useState	useRef
値変更時の再レンダリング	する	しない
用途	UIに表示する値	DOM参照、タイマーID、内部フラグ
値の読み取り	直接	.current プロパティ経由

useContext - コンテキスト消費フック

Contextから現在の値を取得するフック。Providerで包まれた子孫コンポーネントで使用。

💡 初学者向け解説:

通常、データを子コンポーネントに渡すには props を使う。しかし、深くネストしたコンポーネントにデータを渡すには、途中のすべてのコンポーネントを経由する必要がある（「props drilling」問題）。

```
// ✗ props drilling: 中間コンポーネントがthemeを使わないのに渡している
function App() {
  const theme = 'dark'
  return <Header theme={theme} />
}

function Header({ theme }) {
  return <Nav theme={theme} />
}

function Nav({ theme }) {
  return <Button theme={theme} />
}

function Button({ theme }) {
  return <button className={theme}>Click</button>
}
```

Contextを使うと、途中を飛ばして直接データを受け取れる。

```
// ✓ Context: どの階層からでも直接アクセス
const ThemeContext = createContext('light')

function App() {
  return (
    <ThemeContext.Provider value="dark">
      <Header />
    </ThemeContext.Provider>
  )
}

function Header() { return <Nav /> }
function Nav() { return <Button /> }
function Button() {
  const theme = useContext(ThemeContext) // 直接取得！
  return <button className={theme}>Click</button>
}
```

シグネチャ:

```
const value = useContext<T>(Context: React.Context<T>): T
```

構文の読み方:

- Context → createContext で作成したContextオブジェクト
- 戻り値 → 最も近い祖先のProviderが提供している値

本プロジェクトでの使用例:

```
// AdminAuthContext.tsx - カスタムフックでラップ
export function useAdminAuth() {
  const context = useContext(AdminAuthContext)
  if (!context) {
    throw new Error('useAdminAuth must be used within AdminAuthProvider')
  }
  return context
}

// 各コンポーネントでの使用
const { isAdmin, idToken, loginWithGoogle, logout } = useAdminAuth()

if (!isAdmin) {
  return <div>管理者ログインが必要です</div>
}
```

createContext - コンテキスト作成

コンポーネントツリー全体で値を共有するためのContextを作成。

💡 初学者向け解説:

createContext と useContext はセットで使う。Contextは3つのパートで構成される：

1. **Context作成** - createContext(初期値) でContextを作る
2. **Provider** - データを提供する側。value propsで値を渡す
3. **Consumer** - データを使う側。useContext で値を取得

```
// 1. Context作成
const UserContext = createContext<User | null>(null)

// 2. Provider: 値を提供
function App() {
  const [user, setUser] = useState<User | null>(null)

  return (
    <UserContext.Provider value={user}>
      <MainContent />
    </UserContext.Provider>
  )
}

// 3. Consumer: 値を使用
function Profile() {
  const user = useContext(UserContext)
  return <div>{user?.name}</div>
}
```

本プロジェクトでのパターン:

本プロジェクトでは、ContextとProviderをセットでエクスポートし、さらにカスタムフック（useAdminAuth）で使いやすくしている。

```

// contexts/AdminAuthContext.tsx

// 1. Context作成
const AdminAuthContext = createContext<AdminAuthContextType | null>(null)

// 2. Providerコンポーネント
export function AdminAuthProvider({ children }) {
  const [user, setUser] = useState(null)
  // ... ロジック
  return (
    <AdminAuthContext.Provider value={{ user, ... }}>
      {children}
    </AdminAuthContext.Provider>
  )
}

// 3. カスタムフック (useContextをラップ)
export function useAdminAuth() {
  const context = useContext(AdminAuthContext)
  if (!context) throw new Error('Provider内で使ってください')
  return context
}

// 使用例
const { user, loginWithGoogle } = useAdminAuth()

```

シグネチャ:

```
const MyContext = createContext<T>(defaultValue: T): Context<T>
```

構文の読み方:

- defaultValue → Providerがない場合に使われる初期値（通常はnull）
- 戻り値 → Provider と Consumer を持つContextオブジェクト

本プロジェクトでの使用例:

```

// AdminAuthContext.tsx
type AdminAuthContextType = {
  isAdmin: boolean
  user: User | null
  idToken: string | null
  isLoading: boolean
  sessionExpiresAt: number | null
  loginWithGoogle: () => Promise<boolean>
  logout: () => Promise<void>
  registerBeforeLogout: (callback: BeforeLogoutCallback) => () => void
}

// nullを初期値とし、Providerで実際の値を提供
const AdminAuthContext = createContext<AdminAuthContextType | null>(null)

// Provider コンポーネント
export function AdminAuthProvider({ children }: { children: ReactNode }) {
  // ...状態とロジック

  return (
    <AdminAuthContext.Provider value={{{
      isAdmin, user, idToken, isLoading,
      sessionExpiresAt, loginWithGoogle, logout, registerBeforeLogout
    }}}>
      {children}
    </AdminAuthContext.Provider>
  )
}

// main.tsx - アプリ全体をProviderで包む
<BrowserRouter>
  <AdminAuthProvider>
    <GlobalBackground />
    <ScrollTopHomeSwitch />
    <AnimatedRoutes />
  </AdminAuthProvider>
</BrowserRouter>

```

lazy - 遅延読み込み

コンポーネントを動的にインポートし、コード分割を実現。初期バンドルサイズを削減。

💡 初学者向け解説:

通常の `import` はページロード時にすべてのコードを読み込む。しかし、大きなアプリでは使わないページのコードまで読み込むのは無駄。

```
// ✗ 通常のimport: 最初に全部読み込む
import Home from './routes/Home'
import Posts from './routes/Posts'
import AdminPanel from './routes/AdminPanel' // 管理者しか使わないので全員が読み込む
```

lazy を使うと、そのコンポーネントが必要になった時に初めて読み込む（遅延読み込み）。

```
// ✓ lazy: 必要になったら読み込む
const Home = lazy(() => import('./routes/Home'))
const Posts = lazy(() => import('./routes/Posts'))
const AdminPanel = lazy(() => import('./routes/AdminPanel')) // 管理者がアクセスした時だけ読み込む
```

メリット:

- 初期表示が速くなる（最初に読み込むコード量が減る）
- ユーザーが使わない機能のコードはダウンロードされない

注意: lazy コンポーネントは Suspense で囲む必要がある（ローディング中のUIを指定）

シグネチャ:

```
const LazyComponent = lazy(() => import('./Component')): React.LazyExoticComponent
```

構文の読み方:

- `() => import('./Component')` → 動的インポートを返す関数
- `import()` → 実行時にモジュールを読み込むPromiseを返す

本プロジェクトでの使用例:

```
// AnimatedRoutes.tsx - 全ルートの遅延読み込み
const loadHome = () => import(/* webpackPreload: true */ '../routes/Home')
const loadPosts = () => import(/* webpackPreload: true */ '../routes/Posts')
const loadPostDetail = () => import(/* webpackPrefetch: true */ '../routes/PostDetail')
// ...

const Home = lazy(loadHome)
const Posts = lazy(loadPosts)
const PostDetail = lazy(loadPostDetail)
// ...
```

Webpackマジックコメント:

- `webpackPreload`: 高優先度で並行ロード（メインバンドルと同時）
- `webpackPrefetch`: 低優先度でプリフェッチ（ブラウザのアイドル時間に）
- `webpackChunkName`: チャンク名を指定（キャッシュ効率向上）

Suspense - サスペンス境界

遅延読み込みコンポーネントのローディング中にフォールバックUIを表示。

💡 初学者向け解説:

`lazy` でコンポーネントを遅延読み込みすると、読み込み完了まで少し時間がかかる。その間、何を表示するかを指定するのが `Suspense`。

```
// lazyコンポーネントをSuspenseで囲む
<Suspense fallback={<div>読み込み中...</div>}>
  <LazyComponent /> /* 読み込み完了まで fallback が表示される */
</Suspense>
```

fallbackに指定できるもの:

```
// パターン1: テキスト
<Suspense fallback={<p>Loading...</p>}>

// パターン2: スピナー (くるくる回るアイコン)
<Suspense fallback={<Spinner />}>

// パターン3: スケルトン (レイアウトを維持した灰色ボックス)
<Suspense fallback={<PageSkeleton />}>

// パターン4: 何も表示しない (本プロジェクトで採用)
<Suspense fallback={null}>
```

本プロジェクトでは `fallback={null}` を使用。これは既存のUIを維持したまま、新しいコンポーネントが読み込まれるのを待つ。

シグネチャ:

```
<Suspense fallback={<Loading />}>
  <LazyComponent />
</Suspense>
```

構文の読み方:

- `fallback` → ローディング中に表示するJSX
- 子要素 → 読み込み完了後に表示されるコンポーネント

本プロジェクトでの使用例:

```
// AnimatedRoutes.tsx
<Suspense fallback={null}> {/* ローディング中は何も表示しない */}
  <Routes location={location}>
    <Route path="/" element={<App />} />
    <Route path="/home" element={<Home />} />
    <Route path="/posts" element={<Posts />} />
    {/* ... */}
  </Routes>
</Suspense>
```

fallbackの選択肢:

- null : 何も表示しない (既存UIを維持)
- スピナー/スケルトン: ローディングインジケーター
- プレースホルダー: レイアウトを維持するダミーUI

createRef - ref作成

クラス外で使用可能なrefオブジェクトを作成。関数コンポーネント内では通常 useRef を使用。

シグネチャ:

```
const ref = createRef<T>(): RefObject<T>
```

本プロジェクトでの使用例:

現状のコードベースでは createRef は未使用です (以前はページ遷移アニメーションのために使用していました)。

useRefとcreateRefの違い:

特性	useRef	createRef
使用場所	関数コンポーネント内	どこでも
再レンダリング時	同じrefを維持	新しいrefを作成
用途	通常のref用途	動的なref生成

useLayoutEffect - 同期的副作用フック

DOMの変更後、ブラウザの描画前に同期的に実行。レイアウト計算やDOM測定に使用。

💡 初学者向け解説:

useEffect と似ているが、実行タイミングが異なる。

レンダリング完了 → `useLayoutEffect`実行 → 画面に描画 → `useEffect`実行
↑ ここで実行 ↑ ユーザーに見える

なぜ必要？

`useEffect` は画面描画後に実行されるため、DOM操作をすると「一瞬古い状態が見える」ことがある（ちらつき）。

```
// ✗ useEffect: らつきが発生する可能性
function Tooltip() {
  const [position, setPosition] = useState({ top: 0, left: 0 })
  const ref = useRef<HTMLDivElement>(null)

  useEffect(() => {
    // 画面描画後に位置を計算 → 一瞬ずれた位置に表示される
    const rect = ref.current?.getBoundingClientRect()
    setPosition({ top: rect?.top ?? 0, left: rect?.left ?? 0 })
  }, [])

  return <div ref={ref} style={position}>Tooltip</div>
}

// ✓ useLayoutEffect: らつきなし
function Tooltip() {
  const [position, setPosition] = useState({ top: 0, left: 0 })
  const ref = useRef<HTMLDivElement>(null)

  useLayoutEffect(() => {
    // 画面描画前に位置を計算 → 最初から正しい位置に表示
    const rect = ref.current?.getBoundingClientRect()
    setPosition({ top: rect?.top ?? 0, left: rect?.left ?? 0 })
  }, [])

  return <div ref={ref} style={position}>Tooltip</div>
}
```

使い分け:

- **`useEffect` (99%のケース)** : API呼び出し、イベント登録、ログ出力など
- **`useLayoutEffect` (レアケース)** : DOM測定、ちらつき防止、レイアウト調整

シグネチャ:

```
useLayoutEffect(effect: () => void | (()) => void), deps?: DependencyList)
```

構文の読み方:

- `useEffect` と同じ書き方
- 違いは実行タイミングのみ

useEffectとの違い:

特性	useEffect	useLayoutEffect
実行タイミング	描画後（非同期）	描画前（同期）
UIブロック	しない	する（短時間）
用途	データフェッチ、イベントリスナー	DOM測定、レイアウト調整

本プロジェクトでの使用例:

現状のコードベースでは `useLayoutEffect` は未使用です（以前はページ遷移アニメーションのために使用していました）。

useLayoutEffectを使うべき場面:

- DOM要素のサイズ・位置を測定して即座に反映する時
- フリッカーを防ぎたい時（`useEffect`だと一瞬古い状態が見える）
- CSSクラスを遷移開始時に確実に付与したい時

startTransition - 低優先度更新マーク

状態更新を低優先度としてマークし、より重要な更新（ユーザー入力等）を優先。React 18のConcurrent Modeの機能。

💡 初学者向け解説:

通常、Reactの状態更新は「すぐに画面に反映」される。しかし、重い処理が入ると画面がカクついたり、ユーザー入力が遅延したりする。

```
// ✗ 重い処理がユーザー入力をブロック
function Search() {
  const [query, setQuery] = useState('')
  const [results, setResults] = useState([])

  const handleChange = (e) => {
    setQuery(e.target.value)           // 入力を即座に反映したい
    setResults(searchItems(query))   // 重い処理...入力が遅延する
  }

  return <input value={query} onChange={handleChange} />
}
```

`startTransition` を使うと、「この更新は後回しでOK」とReactに伝えられる。

```

// ✓ 重い処理を低優先度に
function Search() {
  const [query, setQuery] = useState('')
  const [results, setResults] = useState([])

  const handleChange = (e) => {
    setQuery(e.target.value) // 高優先度: 即座に反映

    startTransition(() => {
      // 低優先度: ユーザー入力を邪魔しない
      setResults(searchItems(e.target.value))
    })
  }

  return <input value={query} onChange={handleChange} />
}

```

具体的な効果:

- 入力フィールドは即座に反映される（カクつかない）
- 検索結果は少し遅れて更新される（でも体感は良い）

使いどころ:

- フィルタリング/検索
- 大きなリストの更新
- 重いレンダリング処理

シグネチャ:

```
startTransition(callback: () => void): void
```

構文の読み方:

- callback → 低優先度にしたい状態更新を含む関数
- 戻り値なし

本プロジェクトでの使用例:

```

// Posts.tsx - タグフィルタリングの低優先度更新
const handleTagClick = useCallback((tag: string) => {
  startTransition(() => {
    // この更新は低優先度として扱われる
    // ユーザーの他の操作をブロックしない
    setSearchParams(tag ? { tag } : {})
  })
}, [setSearchParams])

```

```
// PostDetail.tsx - いいね数の更新
const handleGood = useCallback(async () => {
  // 楽観的更新（即座にUI反映）
  startTransition(() => {
    setGoodCount((prev) => prev + 1)
    setHasVoted(true)
  })
  // APIコール...
}, /* deps */)
```

startTransitionを使うべき場面:

- フィルタリング、検索結果の更新
- 大量のリスト再レンダリング
- ユーザー体験に影響しない背景更新

ReactNode - React描画可能型

Reactがレンダリングできるすべての型を表す。コンポーネントのchildren型として使用。

💡 初学者向け解説:

Reactでは、コンポーネントの中に「子要素」を渡せる。この子要素の型が `ReactNode`。

```
// 子要素の例
<Container>
  <h1>タイトル</h1>          {/* ReactElement */}
  テキストも書ける          {/* string */}
  {42}                      {/* number */}
  {items.map(i => <li>i</li>)}  {/* ReactNode[] */}
  {null}                     {/* null (何も表示しない) */}
  {isVisible && <Modal />}  {/* boolean | ReactElement */}
</Container>
```

これらすべてを受け入れられる型が `ReactNode`。

コンポーネントでchildrenを受け取る:

```

// 基本的なラッパーコンポーネント
type ContainerProps = {
  children: ReactNode // 何でも受け取れる
}

function Container({ children }: ContainerProps) {
  return <div className="container">{children}</div>
}

// オプショナルな場合は?をつける
type CardProps = {
  title: string
  children?: ReactNode // なくてもOK
}

```

型定義:

```

type ReactNode =
  | ReactElement // <Component /> や <div> など
  | string // "テキスト"
  | number // 42
  | Iterable<ReactNode> // 配列など
  | ReactPortal // createPortalの戻り値
  | boolean // true/false (表示されない)
  | null // 何も表示しない
  | undefined // 何も表示しない

```

本プロジェクトでの使用例:

```

// AdminAuthContext.tsx - Providerのchildren型
export function AdminAuthProvider({ children }: { children: ReactNode }) {
  // ...
  return (
    <AdminAuthContext.Provider value={/* ... */}>
      {children}
    </AdminAuthContext.Provider>
  )
}

```

```
// Lightbox.tsx - オプショナルなchildren
type LightboxProps = {
  isOpen: boolean
  onClose: () => void
  imageSrc: string
  imageAlt: string
  children?: ReactNode // 画像下に追加情報を表示
}
```

react-dom

ReactをDOM（ブラウザ）環境/サーバー環境でレンダリングするためのライブラリ。

インポートされるAPI:

API	説明	使用ファイル
createRoot	CSR時にアプリをDOMにマウント	main.tsx
hydrateRoot	SSG/SSRで事前生成されたHTMLをハイドレート	main.tsx
renderToString	SSG/SSR用にHTML文字列を生成	entry-server.tsx

// 使用例

```
hydrateRoot(rootElement, app)
const html = renderToString(<App />)
```

react-router-dom

React用のクライアントサイドルーティングライブラリ。SPAでのページ遷移を管理。

インポートされるAPI:

API	説明	使用ファイル例
BrowserRouter	HTML5 History APIを使用したルーター	main.tsx
Routes	ルート定義のコンテナ	AnimatedRoutes.tsx
Route	個別のルート定義	AnimatedRoutes.tsx
Link	クライアントサイドナビゲーション用リンク	全ページコンポーネント
useNavigate	プログラマティックなナビゲーション用フック	App.tsx , ScrollTopHomeSwitch.tsx , BackButton.tsx

API	説明	使用ファイル例
useLocation	現在のURL情報を取得	AnimatedRoutes.tsx , GlobalBackground.tsx
useParams	URLパラメータを取得	PostDetail.tsx , BBSThread.tsx
useSearchParams	クエリパラメータを取得・設定	Posts.tsx
LinkProps	Linkコンポーネントのprops型	PrefetchLink.tsx

```
// 使用例
const navigate = useNavigate()
navigate('/home') // /homeへ遷移

const { slug } = useParams<{ slug: string }>() // URL: /posts/:slug
```

react-router

SSG/SSR向けのルーティング（`StaticRouter` など）を提供。 `react-router-dom` の内部依存としても利用されます。

インポートされるAPI:

API	説明	使用ファイル
StaticRouter	サーバー側でlocationを固定してレンダリング	entry-server.tsx

```
// 使用例
<StaticRouter location="/posts/example">
  <App />
</StaticRouter>
```

p5

クリエイティブコーディング向けの描画ライブラリ。背景アニメ（`P5HypercubeBackground.tsx`）でWEBGL描画に使用します（動的importで必要時のみロード）。

Firebase関連

firebase

Googleが提供するBaaS（Backend as a Service）。認証、データベース、ストレージ等を提供。

インポートされるモジュール:

`firebase/app`

API	説明	使用ファイル
<code>initializeApp</code>	Firebaseアプリを初期化	<code>firebase.ts</code>

`firebase/auth`

API	説明	使用ファイル
<code>getAuth</code>	Authインスタンスを取得	<code>firebase.ts</code>
<code>GoogleAuthProvider</code>	Google認証プロバイダー	<code>firebase.ts</code>
<code>signInWithPopup</code>	ポップアップでソーシャルログイン	<code>AdminAuthContext.tsx</code>
<code>signOut</code>	ログアウト	<code>AdminAuthContext.tsx</code>
<code>onAuthStateChanged</code>	認証状態の変更を監視	<code>AdminAuthContext.tsx</code>
<code>User</code>	Firebaseユーザー型	<code>AdminAuthContext.tsx</code>

```
// 使用例
const app = initializeApp(firebaseConfig)
const auth = getAuth(app)
const provider = new GoogleAuthProvider()

// ログイン
const result = await signInWithPopup(auth, provider)
const user = result.user
const token = await user.getIdToken()

// 認証状態監視
onAuthStateChanged(auth, (user) => {
  if (user) { /* ログイン中 */ }
})
```

Markdown処理関連

`unified`

テキスト処理のためのプラグインベースのエコシステム。AST（抽象構文木）を操作してテキストを変換。

```
// 使用例: Markdown → HTML変換パイプライン
const result = await unified()
  .use(remarkParse)      // Markdown → mdast
  .use(remarkGfm)        // GFM拡張
  .use(remarkRehype)     // mdast → hast
  .use(rehypeStringify)  // hast → HTML文字列
  .process(markdown)
```

remark-parse

Markdownテキストをmdast (Markdown AST) にパース。

remark-gfm

GitHub Flavored Markdown (GFM) をサポート。テーブル、取り消し線、タスクリスト、URL自動リンク等。

remark-rehype

mdast (Markdown AST) をhast (HTML AST) に変換。

rehype-stringify

hast (HTML AST) をHTML文字列にシリアル化。

remark-math

Markdownで数式記法 ($\$...$$, $\$\$\dots\$\$$) をサポート。

rehype-katex

hast内の数式ノードをKaTeXでレンダリング。

katex

高速な数式レンダリングライブラリ。LaTeX記法をHTMLに変換。

```
// 使用例 (MarkdownEditor.tsx)
<MDEditor.Markdown
  source={content}
  remarkPlugins={[remarkMath]}
  rehypePlugins={[rehypeKatex]}
/>
```

Markdownエディタ

@uiw/react-md-editor

React用のMarkdownエディタコンポーネント。リアルタイムプレビュー、ツールバー、カスタマイズ可能。

インポートされるAPI:

API	説明	使用ファイル
MDEditor	メインのエディタコンポーネント	MarkdownEditor.tsx
MDEditor.Markdown	Markdownプレビューコンポーネント	MarkdownEditor.tsx
commands	ツールバーコマンド（太字、イタリック等）	MarkdownEditor.tsx

```
// 使用例
<MDEditor
  value={value}
  onChange={(val) => onChange(val || '')}
  height={500}
  preview="live"
  commands={[
    commands.bold,
    commands.italic,
    commands.link,
    commands.code,
    // ...
  ]}
/>
```

ダイアグラム描画

`mermaid`

テキストベースのダイアグラム描画ライブラリ。フローチャート、シーケンス図、クラス図等を生成。

インポートされるAPI:

API	説明	使用ファイル
<code>mermaid.initialize</code>	Mermaidの初期設定（テーマ等）	<code>PostDetail.tsx</code> , <code>MermaidRenderer.tsx</code>
<code>mermaid.render</code>	Mermaidコードを非同期でSVGにレンダリング	<code>MermaidRenderer.tsx</code>

```
// 使用例
mermaid.initialize({
  startOnLoad: false,
  theme: 'dark',
  themeVariables: {
    primaryColor: '#4f46e5',
    // ...
  },
})

const { svg } = await mermaid.render('diagram-id', `
graph TD
A[開始] --> B[処理]
B --> C[終了]
`)
`)
```

ファイル別インポート一覧

src/main.tsx

```
import { BrowserRouter } from 'react-router-dom'
import { createRoot, hydrateRoot } from 'react-dom/client'
import AnimatedRoutes from './components/AnimatedRoutes'
import GlobalBackground from './components/GlobalBackground'
import ScrollTopHomeSwitch from './components/ScrollTopHomeSwitch'
import { AdminAuthProvider } from './contexts/AdminAuthContext'
import './index.css'
```

src/entry-server.tsx

```
import { renderToString } from 'react-dom/server'
import { StaticRouter } from 'react-router'
import { AdminAuthProvider } from './contexts/AdminAuthContext'
import GlobalBackground from './components/GlobalBackground'
import ScrollTopHomeSwitch from './components/ScrollTopHomeSwitch'
import ServerRoutes from './components/ServerRoutes'
```

src/App.tsx

```
import { useEffect, useRef, useCallback } from 'react'
import { useNavigate } from 'react-router-dom'
```

src-contexts/AdminAuthContext.tsx

```
import { createContext, useContext, useState, useCallback, useEffect, useRef, type ReactNode } from 'react'
import { signInWithPopup, signOut, onAuthStateChanged, type User } from 'firebase/auth'
import { auth, googleProvider } from '../lib/firebase'
```

src-components/AnimatedRoutes.tsx

```
import { Routes, Route, useLocation } from 'react-router-dom'
import { lazy, Suspense, useEffect } from 'react'
import App from '../App'
import { shouldPrefetch } from '../lib/network'
import { preload, prefetch, lazyLoad } from '../lib/preload'
```

src-components/ServerRoutes.tsx

```
import { Routes, Route, useLocation } from 'react-router-dom'
import App from '../App'
import Home from '../routes/Home'
import Posts from '../routes/Posts'
import PostDetail from '../routes/PostDetail'
import Products from '../routes/Products'
import ProductDetail from '../routes/ProductDetail'
import Photos from '../routes/Photos'
import BBSList from '../routes/BBSList'
import BBSThread from '../routes/BBSThread'
```

src-components/GlobalBackground.tsx

```
import { useLocation } from 'react-router-dom'
import P5HypercubeBackground from './P5HypercubeBackground'
```

src-components/P5HypercubeBackground.tsx

```
import { useEffect, useRef } from 'react'
```

src-components/ScrollTopHomeSwitch.tsx

```
import { useEffect, useRef } from 'react'
import { useLocation, useNavigate } from 'react-router-dom'
```

src-components/AccessCounter.tsx

```
import { useEffect, useRef, useState } from 'react'
```

src/components/ClientOnly.tsx

```
import { useEffect, useState, type ReactNode } from 'react'
```

src/components/SiteFooter.tsx

```
import AccessCounter from './AccessCounter'  
import ClientOnly from './ClientOnly'  
import { MAIN_TEXT_STYLE } from '../styles/typography'
```

src/components/PrefetchLink.tsx

```
import { Link, type LinkProps } from 'react-router-dom'  
import { useCallback, useRef } from 'react'
```

src/components/Lightbox.tsx

```
import { useEffect } from 'react'
```

src/components/MermaidRenderer.tsx

```
import { useEffect, useRef, useState } from 'react'  
import mermaid from 'mermaid'
```

src/components/BackButton.tsx

```
import { useCallback } from 'react'  
import { useNavigate } from 'react-router-dom'
```

src/components/admin/MarkdownEditor.tsx

```
import { useState, useCallback, useRef, useEffect, useMemo } from 'react'  
import MDEditor, { commands } from '@uiw/react-md-editor'  
import { useAdminAuth } from '../../contexts/AdminAuthContext'  
import rehypeKatex from 'rehype-katex'  
import remarkMath from 'remark-math'  
import mermaid from 'mermaid'  
import 'katex/dist/katex.min.css'
```

src/routes/Home.tsx

```
import { useEffect, useRef, useState } from 'react'
import PrefetchLink from '../components/PrefetchLink'
import SiteFooter from '../components/SiteFooter'
import postsData from '../data/posts.json' with { type: 'json' }
import { usePageMeta } from '../hooks/usePageMeta'
import { useReveal } from '../hooks/useReveal'
import { useScrollToTop } from '../hooks/useScrollToTop'
import { CMS_ENDPOINT } from '../lib/endpoints'
import { MAIN_TEXT_STYLE } from '../styles/typography'
```

src/routes/Posts.tsx

```
import { useSearchParams, Link } from 'react-router-dom'
import { useEffect, useMemo, useRef, useCallback, startTransition, useState } from 'react'
import postsData from '../data/posts.json' with { type: 'json' }
import PrefetchLink from '../components/PrefetchLink'
import SiteFooter from '../components/SiteFooter'
import { useAdminAuth } from '../contexts/AdminAuthContext'
import { useReveal } from '../hooks/useReveal'
import { useScrollToTop } from '../hooks/useScrollToTop'
import { formatDraftDate } from '../lib/draftStorage'
import { CMS_ENDPOINT } from '../lib/endpoints'
import { usePageMeta } from '../hooks/usePageMeta'
import { MAIN_FONT_STYLE, MAIN_TEXT_STYLE } from '../styles/typography'
```

src/routes/PostDetail.tsx

```
import { useLocation, useParams, Link } from 'react-router-dom'
import { useEffect, useMemo, useRef, useState, useCallback, startTransition } from 'react'
import postsData from '../data/posts.json' with { type: 'json' }
import PrefetchLink from '../components/PrefetchLink'
import SiteFooter from '../components/SiteFooter'
import ClientOnly from '../components/ClientOnly'
import { useMermaidBlocks } from '../hooks/useMermaidBlocks'
import { useReveal } from '../hooks/useReveal'
import { useScrollToTop } from '../hooks/useScrollToTop'
import { CMS_ENDPOINT, GOOD_ENDPOINT } from '../lib/endpoints'
import { usePageMeta } from '../hooks/usePageMeta'
import { MAIN_FONT_STYLE, MAIN_TEXT_STYLE } from '../styles/typography'
```

src/routes/Products.tsx

```
import { useEffect, useRef, useState } from 'react'
import { Link } from 'react-router-dom'

import productsData from '../data/products.json' with { type: 'json' }
import ArrowRightIcon from '../components/icons/ArrowRightIcon'
import PrefetchLink from '../components/PrefetchLink'
import SiteFooter from '../components/SiteFooter'
import { useAdminAuth } from '../contexts/AdminAuthContext'
import { usePageMeta } from '../hooks/usePageMeta'
import { useReveal } from '../hooks/useReveal'
import { useScrollToTop } from '../hooks/useScrollToTop'
import { CMS_ENDPOINT } from '../lib/endpoints'
import { MAIN_FONT_STYLE, MAIN_TEXT_STYLE } from '../styles/typography'
```

src/routes/ProductDetail.tsx

```
import { useEffect, useRef } from 'react'
import { useParams, Link } from 'react-router-dom'

import productsData from '../data/products.json' with { type: 'json' }
import productPostsData from '../data/product-posts.json' with { type: 'json' }
import PrefetchLink from '../components/PrefetchLink'
import SiteFooter from '../components/SiteFooter'
import { useMermaidBlocks } from '../hooks/useMermaidBlocks'
import { usePageMeta } from '../hooks/usePageMeta'
import { useReveal } from '../hooks/useReveal'
import { useScrollToTop } from '../hooks/useScrollToTop'
import { MAIN_FONT_STYLE, MAIN_TEXT_STYLE } from '../styles/typography'
```

src/routes/Photos.tsx

```
import Lightbox from '../components/Lightbox'
import PrefetchLink from '../components/PrefetchLink'
import SiteFooter from '../components/SiteFooter'
import { useState, useRef } from 'react'
import { photos, shotTags, type Photo, type PhotoRatio } from '../data/photos'
import { usePageMeta } from '../hooks/usePageMeta'
import { useReveal } from '../hooks/useReveal'
import { useScrollToTop } from '../hooks/useScrollToTop'
import { MAIN_FONT_STYLE, MAIN_TEXT_STYLE } from '../styles/typography'
```

src/routes/BBSList.tsx

```
import { useEffect, useRef, useState, useCallback } from 'react'
import { Link } from 'react-router-dom'
import PrefetchLink from '../components/PrefetchLink'
import SiteFooter from '../components/SiteFooter'
import { useAdminAuth } from '../contexts/AdminAuthContext'
import { usePageMeta } from '../hooks/usePageMeta'
import { useReveal } from '../hooks/useReveal'
import { useScrollToTop } from '../hooks/useScrollToTop'
import { BBS_ENDPOINT } from '../lib/endpoints'
import { MAIN_FONT_STYLE, MAIN_TEXT_STYLE } from '../styles/typography'
```

src/routes/BBSThread.tsx

```
import { useEffect, useRef, useState, useCallback } from 'react'
import { useParams, Link } from 'react-router-dom'
import PrefetchLink from '../components/PrefetchLink'
import SiteFooter from '../components/SiteFooter'
import { useAdminAuth } from '../contexts/AdminAuthContext'
import { usePageMeta } from '../hooks/usePageMeta'
import { useReveal } from '../hooks/useReveal'
import { useScrollToTop } from '../hooks/useScrollToTop'
import { BBS_ENDPOINT } from '../lib/endpoints'
import { MAIN_FONT_STYLE, MAIN_TEXT_STYLE } from '../styles/typography'
```

src/routes/admin/PostEditor.tsx

```
import { useEffect, useState, useCallback, useRef } from 'react'
import { useParams, useNavigate, Link } from 'react-router-dom'
import { useAdminAuth } from '../../contexts/AdminAuthContext'
import MarkdownEditor from '../../../../../components/admin/MarkdownEditor'
import { unified } from 'unified'
import remarkParse from 'remark-parse'
import remarkGfm from 'remark-gfm'
import remarkRehype from 'remark-rehype'
import rehypeStringify from 'rehype-stringify'
import { saveDraft, loadDraft, deleteDraft, formatDraftDate, type PostDraft } from '../../../../../lib/draftStorage'
```

src/routes/admin/ProductEditor.tsx

```
import { useEffect, useState, useCallback, useRef } from 'react'
import { useParams, useNavigate, Link } from 'react-router-dom'
import { useAdminAuth } from '../../contexts/AdminAuthContext'
import MarkdownEditor from '../../components/admin/MarkdownEditor'
import { unified } from 'unified'
import remarkParse from 'remark-parse'
import remarkGfm from 'remark-gfm'
import remarkRehype from 'remark-rehype'
import rehypeStringify from 'rehype-stringify'
import { saveDraft, loadDraft, deleteDraft, formatDraftDate, type ProductDraft } from '../../lib/draftStorage'
```

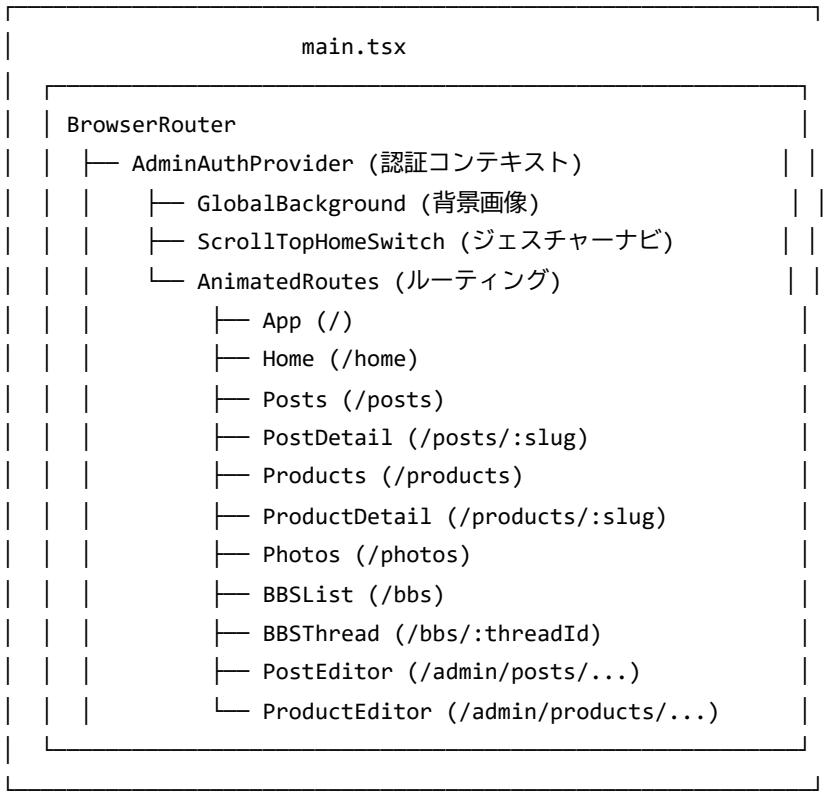
src/lib.firebaseio.ts

```
import { initializeApp } from 'firebase/app'
import { getAuth, GoogleAuthProvider } from 'firebase/auth'
```

環境変数

変数名	説明
VITE_FIREBASE_API_KEY	Firebase APIキー
VITE_FIREBASE_AUTH_DOMAIN	Firebase認証ドメイン
VITE_FIREBASE_PROJECT_ID	FirebaseプロジェクトID
VITE_FIREBASE_STORAGE_BUCKET	Firebaseストレージバケット
VITE_FIREBASE_MESSAGING_SENDER_ID	FirebaseメッセージングID
VITE_FIREBASE_APP_ID	FirebaseアプリID
VITE_ADMIN_EMAILS	管理者メールアドレス（カンマ区切り）
VITE_CMS_ENDPOINT	CMS APIエンドポイント
VITE_GOOD_ENDPOINT	いいねAPIエンドポイント
VITE_BBS_ENDPOINT	BBS APIエンドポイント

アーキテクチャ概要



特記事項

セッション管理

1. ログイン時に1時間のセッションタイムアウトが開始
2. IDトークンは50分ごとに自動更新
3. ページリロード時もセッションは維持（リロード時から1時間）
4. セッション期限切れ時は自動ログアウト & アラート

下書き保存

1. 編集中のデータは`localStorage`に自動保存可能
2. ログアウト時に下書きを自動保存
3. ページ読み込み時に下書きがあれば復元通知
4. サーバーより新しい下書きのみ復元

パフォーマンス最適化

1. 全ルートの遅延読み込み（`React.lazy`）

2. `preload` / `prefetch` による優先度付きロード（アイドル時の事前ロードを含む）
3. ホバー/フォーカス時のルートプリフェッチ（`PrefetchLink`）
4. 背景画像のsrcset対応 + CSS変数による動的ブラー
5. SSG（SSRビルド + prerender）と `hydrateRoot` による初期表示最適化

セキュリティ実装

このセクションでは、haroin57-webで実装しているセキュリティ対策を初学者向けに詳しく解説します。

1. なぜセキュリティが必要なのか

Webアプリケーションでは、「誰がリクエストを送っているか」を確認することが非常に重要です。例えば：

- 記事の編集・削除は管理者だけができるべき
- 下書き記事は一般ユーザーには見せたくない
- 不正なリクエストからAPIを保護したい

これらを実現するために、**認証（Authentication）と認可（Authorization）**という仕組みを使います。

用語	意味	例
認証	「あなたは誰？」を確認する	ログインして本人確認
認可	「あなたはこれをしていい？」を確認する	管理者だけが編集可能

2. JWT (JSON Web Token) とは

2.1 JWTの基本概念

JWTは、ユーザー情報を安全にやり取りするための「デジタル身分証明書」のようなものです。

eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiYWRtaW4iOnRydWUsImhlhd

一見すると意味不明な文字列ですが、実は3つの部分からできています：

[ヘッダー].[ペイロード].[署名]

それぞれ**Base64URL**という形式でエンコード（変換）されています。

2.2 JWTの3つの構成要素

① ヘッダー (Header)

「このトークンは何の方式で署名されているか」を示します。

```
{  
  "alg": "RS256",    // 署名アルゴリズム (RSA + SHA-256)  
  "kid": "abc123",   // 公開鍵のID (Key ID)  
  "typ": "JWT"       // トークンの種類  
}
```

② ペイロード (Payload)

ユーザー情報や有効期限などの「中身」が入っています。

```
{  
  "iss": "https://securetoken.google.com/my-project",  // 発行者  
  "aud": "my-project",                                // 対象者 (誰向けか)  
  "sub": "user123",                                    // ユーザーID  
  "email": "user@example.com",                         // メールアドレス  
  "email_verified": true,                             // メール確認済みか  
  "iat": 1700000000,                                  // 発行時刻  
  "exp": 1700003600,                                  // 有効期限  
  "auth_time": 1699999000                            // 認証した時刻  
}
```

③ 署名 (Signature)

「このトークンが本物であること」を証明する電子署名です。

署名 = RSA暗号化(ヘッダー + " ." + ペイロード, 秘密鍵)

2.3 なぜ署名が必要なのか

署名がなければ、悪意のあるユーザーが自分でトークンを作れてしまいます：

```
// 悪意のあるユーザーが作った偽のペイロード  
{  
  "email": "admin@example.com", // 管理者になりすまし！  
  "email_verified": true  
}
```

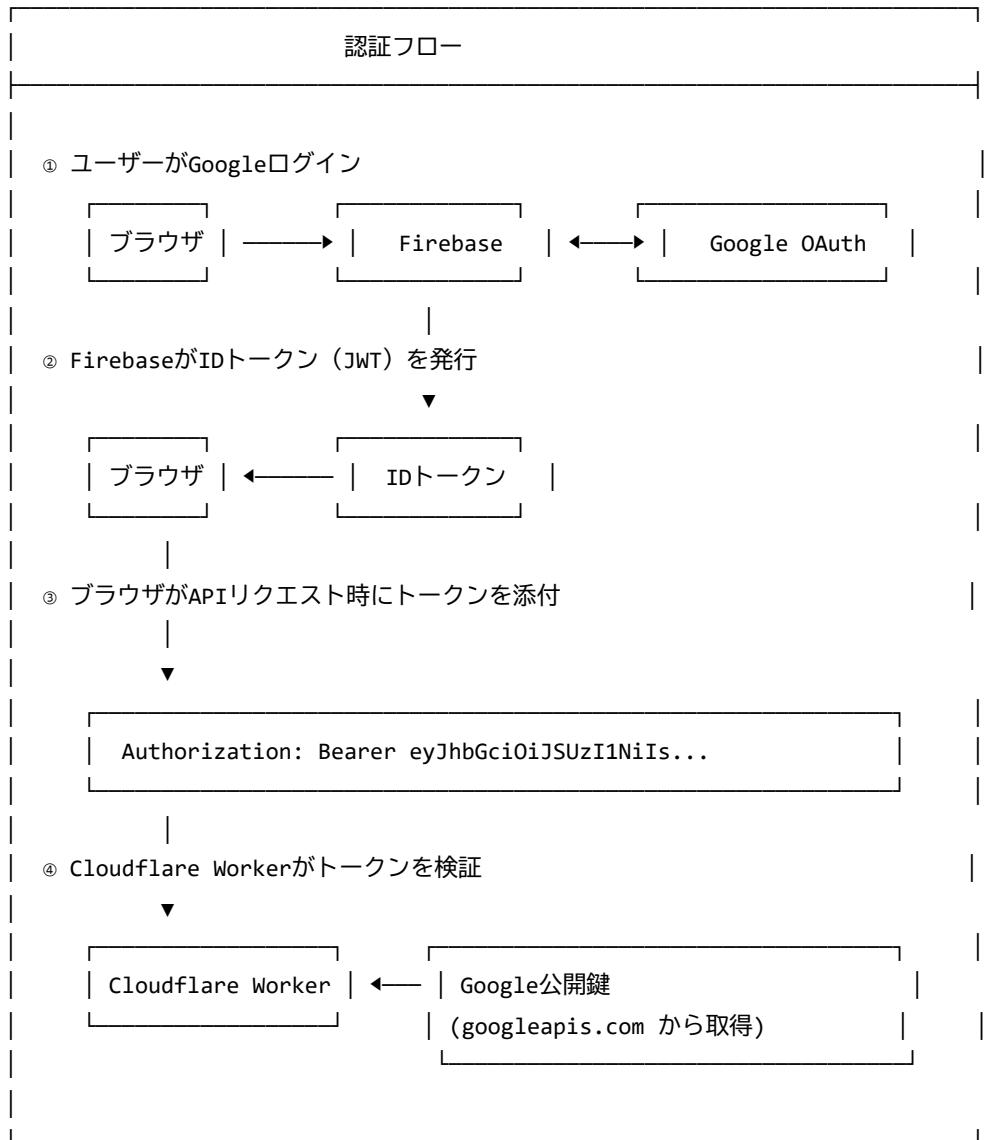
しかし、署名があれば：

1. トークンは**秘密鍵**で署名される (Firebaseだけが持っている)
2. サーバーは**公開鍵**で署名を検証できる
3. 署名が合わなければ偽物と判断できる

これを**公開鍵暗号方式**と呼びます。

3. Firebase認証の仕組み

3.1 認証フロー全体像



3.2 フロントエンド側の実装

`src/contexts/AdminAuthContext.tsx` で認証状態を管理しています：

```

// Googleログイン処理
const loginWithGoogle = async () => {
  const provider = new GoogleAuthProvider()
  const result = await signInWithPopup(auth, provider)

  // FirebaseからIDトークンを取得
  const token = await result.user.getIdToken()
  setIdToken(token)
}

```

APIリクエスト時のトークン送信：

```

// 管理者APIを呼び出す例
const response = await fetch('/api/cms/posts/drafts', {
  headers: {
    'Authorization': `Bearer ${idToken}` // ここでトークンを送る
  }
})

```

`Bearer` は「持参人」という意味で、「このトークンを持っている人を認証してください」という意味になります。

4. サーバー側のトークン検証（詳細解説）

`src/pv-worker.ts` で実装している検証処理を詳しく見ていきます。

4.1 検証の全体像

```

async function verifyFirebaseToken(token: string, env: Env): Promise<FirebaseTokenPayload | null> {
  // 1. トークンを3つの部分に分割
  const parts = token.split('.')
  if (parts.length !== 3) return null // JWTは必ず3部構成

  // 2. ヘッダーの検証
  // 3. ペイロードの検証（各クレーム）
  // 4. 署名の検証

  return payload // 全て通過したら成功
}

```

4.2 各検証項目の詳細

① アルゴリズムの検証（alg）

```
const header = JSON.parse(base64UrlDecode(parts[0]))  
  
// RS256以外は拒否（ダウングレード攻撃防止）  
if (header.alg !== 'RS256') {  
  console.log('Invalid algorithm:', header.alg)  
  return null  
}
```

なぜこれが必要？

→ 攻撃者が alg: "none" を指定して「署名なしでOK」と偽装する攻撃を防ぐため。

② 有効期限の検証 (exp)

```
const now = Math.floor(Date.now() / 1000) // 現在時刻（UNIX時間）  
  
if (!payload.exp || payload.exp < now) {  
  console.log('Token expired')  
  return null  
}
```

exp (expiration) は「このトークンはいつまで有効か」を示します。期限切れトークンは無効です。

③ 発行時刻の検証 (iat)

```
if (!payload.iat || payload.iat > now) {  
  console.log('Invalid iat:', payload.iat)  
  return null  
}
```

iat (issued at) は「トークンがいつ発行されたか」。未来の時刻で発行されたトークンは不正です。

④ 認証時刻の検証 (auth_time)

```
if (!payload.auth_time || payload.auth_time > now) {  
  console.log('Invalid auth_time:', payload.auth_time)  
  return null  
}
```

ユーザーが実際にログインした時刻も過去である必要があります。

⑤ 発行者の検証 (iss)

```
const expectedIssuer = `https://securetoken.google.com/${env.FIREBASE_PROJECT_ID}`
if (payload.iss !== expectedIssuer) {
  console.log('Invalid issuer:', payload.iss)
  return null
}
```

iss (issuer) は「誰がこのトークンを発行したか」。Firebase以外が発行したトークンは拒否します。

⑥ 対象者の検証 (aud)

```
if (payload.aud !== env.FIREBASE_PROJECT_ID) {
  console.log('Invalid audience:', payload.aud)
  return null
}
```

aud (audience) は「このトークンは誰向けか」。自分のプロジェクト向けでないトークンは拒否します。

⑦ ユーザーIDの検証 (sub)

```
if (!payload.sub || typeof payload.sub !== 'string') {
  console.log('Invalid sub')
  return null
}
```

sub (subject) はユーザーの一意識別子。必ず存在する必要があります。

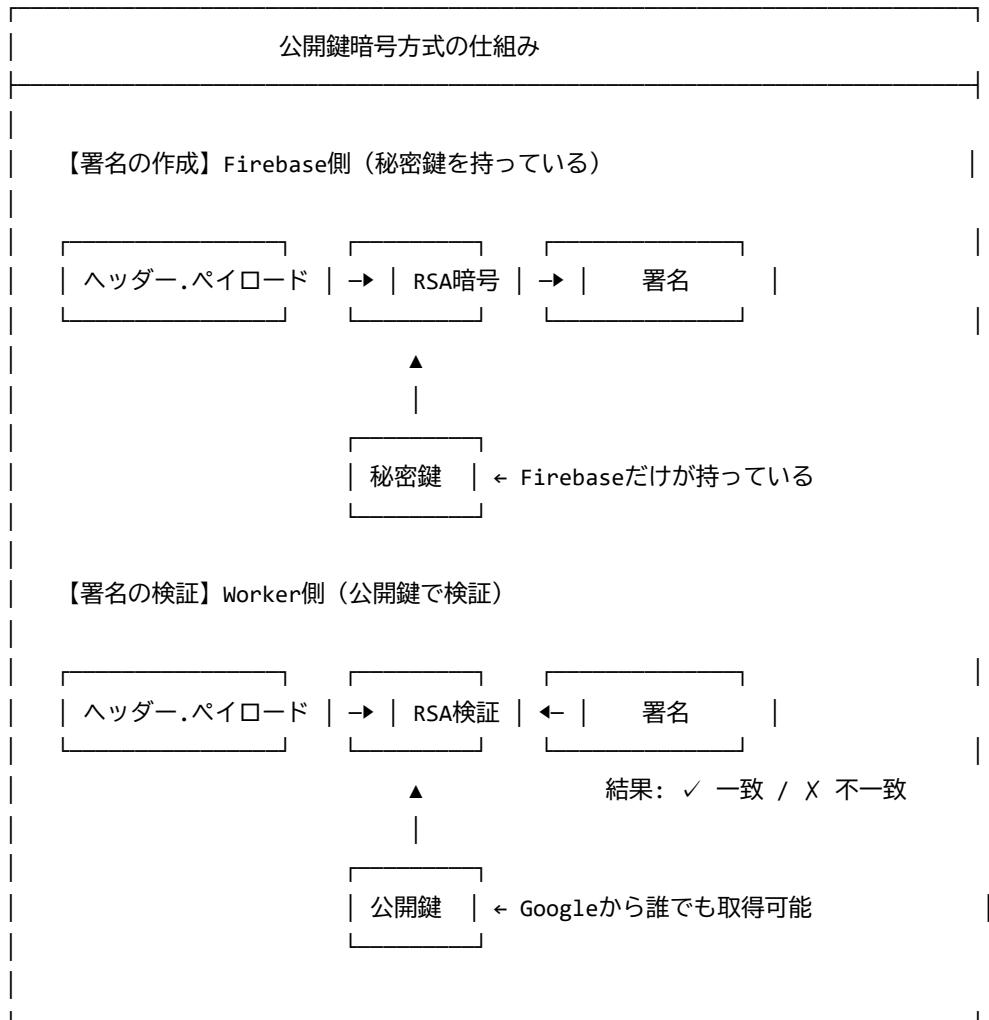
⑧ メール確認済みの検証 (email_verified)

```
if (!payload.email_verified) {
  console.log('Email not verified')
  return null
}
```

メールアドレスが確認されていないユーザーは管理者として認めません。

5. RS256署名検証の仕組み

5.1 公開鍵暗号方式とは



5.2 Google公開鍵の取得

```
const GOOGLE_CERTS_URL = 'https://www.googleapis.com/robot/v1/metadata/x509/securetoken@system.gserviceaccount.com'

async function getGooglePublicKeys(): Promise<Record<string, CryptoKey>> {
    // キャッシュが有効ならそれを使う
    if (publicKeyCache && publicKeyCache.expiresAt > Date.now()) {
        return publicKeyCache.keys
    }

    // Googleから証明書を取得
    const response = await fetch(GOOGLE_CERTS_URL)
    const certs = await response.json()

    // Cache-Controlヘッダーからキャッシュ期間を取得
    const cacheControl = response.headers.get('Cache-Control')
    const maxAge = /* max-ageの値を抽出 */

    // 証明書をCryptoKeyオブジェクトに変換
    for (const [kid, pem] of Object.entries(certs)) {
        keys[kid] = await importPublicKeyFromCert(pem)
    }

    // キャッシュを更新
    publicKeyCache = { keys, expiresAt: Date.now() + maxAge * 1000 }

    return keys
}
```

なぜキャッシュが必要？

- 毎回Googleにアクセスするとレスポンスが遅くなる
- Googleの公開鍵は頻繁には変わらない（数時間～数日）
- Cache-Control ヘッダーの max-age に従ってキャッシュ

5.3 署名検証の実装

```
// 署名検証 (RS256)
const publicKeys = await getGooglePublicKeys()
const publicKey = publicKeys[header.kid] // kidで公開鍵を特定

if (!publicKey) {
  console.log('Public key not found for kid:', header.kid)
  return null
}

// 署名対象データ: ヘッダー.ペイロード
const signedData = new TextEncoder().encode(` ${parts[0]}.${parts[1]}`)

// 署名をBase64URLからバイナリに変換
const signature = base64UrlToArrayBuffer(parts[2])

// WebCrypto APIで検証
const isValid = await crypto.subtle.verify(
  { name: 'RSASSA-PKCS1-v1_5' },
  publicKey,
  signature,
  signedData
)

if (!isValid) {
  console.log('Invalid signature')
  return null
}
```

WebCrypto API はブラウザやCloudflare Workersで使える標準的な暗号化API。外部ライブラリ不要で署名検証ができます。

6. X.509証明書の解析

Googleから取得する公開鍵はPEM形式のX.509証明書として提供されます。

6.1 証明書の形式

```
-----BEGIN CERTIFICATE-----
MIIDJjCCAg6gAwIBAgIIYS... (Base64エンコードされたバイナリ)
-----END CERTIFICATE-----
```

6.2 証明書からの公開鍵抽出

```
async function importPublicKeyFromCert(pem: string): Promise<CryptoKey> {
  // 1. PEMヘッダー/フッターを削除
  const pemContents = pem
    .replace(/-----BEGIN CERTIFICATE-----/g, '')
    .replace(/-----END CERTIFICATE-----/g, '')
    .replace(/\s/g, '')

  // 2. Base64デコードしてバイナリに変換
  const binaryDer = atob(pemContents)
  const bytes = new Uint8Array(binaryDer.length)
  for (let i = 0; i < binaryDer.length; i++) {
    bytes[i] = binaryDer.charCodeAt(i)
  }

  // 3. ASN.1 DER形式の証明書から公開鍵部分（SPKI）を抽出
  const spki = extractSpkiFromCertificate(bytes)

  // 4. WebCrypto APIでCryptoKeyオブジェクトに変換
  return await crypto.subtle.importKey(
    'spki',                                // 形式
    spki,                                    // 公開鍵データ
    { name: 'RSASSA-PKCS1-v1_5', hash: 'SHA-256' }, // アルゴリズム
    false,                                   // エクスポート不可
    ['verify']                               // 検証用途
  )
}
```

6.3 ASN.1とは

ASN.1 (Abstract Syntax Notation One) は、データ構造を定義するための標準規格です。X.509証明書はASN.1のDER (Distinguished Encoding Rules) 形式でエンコードされています。

証明書の構造（簡略化）：

```
SEQUENCE {
    tbsCertificate: SEQUENCE {
        version [0]: INTEGER
        serialNumber: INTEGER
        signature: AlgorithmIdentifier
        issuer: Name
        validity: Validity
        subject: Name
        subjectPublicKeyInfo: SEQUENCE { ← これを抽出
            algorithm: AlgorithmIdentifier
            subjectPublicKey: BIT STRING
        }
    }
    signatureAlgorithm: AlgorithmIdentifier
    signatureValue: BIT STRING
}
```

7. 管理者認可の仕組み

トークンが有効でも、そのユーザーが「管理者かどうか」の確認が別途必要です。

```
async function checkAdminAuth(req: Request, env: Env): Promise<boolean> {
    const authHeader = req.headers.get('Authorization')

    if (authHeader?.startsWith('Bearer ')) {
        const token = authHeader.slice(7) // "Bearer "を除去
        const payload = await verifyFirebaseToken(token, env)

        if (payload?.email) {
            // 環境変数で設定した管理者メールリストと照合
            const adminEmails = (env.ADMIN_EMAILS || '')
                .split(',')
                .map(e => e.trim().toLowerCase())

            if (adminEmails.includes(payload.email.toLowerCase())) {
                return true // 管理者として認証成功
            }
        }
    }

    return false // 管理者ではない
}
```

環境変数の設定例（Cloudflare Dashboard）：

```
ADMIN_EMAILS = "admin1@example.com,admin2@example.com"
FIREBASE_PROJECT_ID = "my-firebase-project"
```

8. セキュリティのベストプラクティス

8.1 実装されている対策一覧

対策	目的	実装箇所
RS256署名検証	トークン偽造防止	verifyFirebaseToken()
アルゴリズム固定 (RS256)	ダウングレード攻撃防止	ヘッダー検証
有効期限チェック (exp)	期限切れトークン拒否	ペイロード検証
発行者検証 (iss)	不正発行元トークン拒否	ペイロード検証
対象者検証 (aud)	他プロジェクト向けトークン拒否	ペイロード検証
公開鍵キャッシュ	パフォーマンス最適化	getGooglePublicKeys()
CORS設定	クロスサイトリクエスト制御	corsHeaders
管理者メールリスト	認可制御	checkAdminAuth()

8.2 やってはいけないこと

```
// ✗ 署名を検証せずにペイロードを信用する
const payload = JSON.parse(atob(token.split('.')[1]))
if (payload.email === 'admin@example.com') {
    // 危険！誰でもこのペイロードを作れる
}

// ✗ アルゴリズムを動的に決める
const alg = header.alg // 攻撃者が "none" を指定できてしまう

// ✗ 有効期限をチェックしない
// 一度発行されたトークンが永久に有効になってしまう
```

8.3 本番環境でのチェックリスト

- FIREBASE_PROJECT_ID が正しく設定されているか
- ADMIN_EMAILS に管理者のメールのみが含まれているか
- HTTPS経由でのみAPIにアクセスできるか
- 本番環境の秘密情報がログに出力されていないか
- 公開鍵のキャッシュが正しく動作しているか

9. トラブルシューティング

9.1 よくあるエラーと対処法

エラーメッセージ	原因	対処法
Token expired	トークンの有効期限切れ	フロントエンドで <code>getIdToken(true)</code> を呼んで再取得
Invalid issuer	<code>FIREBASE_PROJECT_ID</code> の設定ミス	環境変数を確認
Invalid audience	別プロジェクトのトークン	FirebaseプロジェクトIDを確認
Public key not found	Google公開鍵の取得失敗	ネットワーク接続を確認、キャッシュをクリア
Invalid signature	トークンが改ざんされている	正規のログインフローを使用しているか確認

9.2 デバッグ方法

```
// トークンの中身を確認（開発時のみ）
const parts = token.split('.')
console.log('Header:', JSON.parse(atob(parts[0])))
console.log('Payload:', JSON.parse(atob(parts[1])))
```

注意: 本番環境ではトークンをログに出力しないでください。

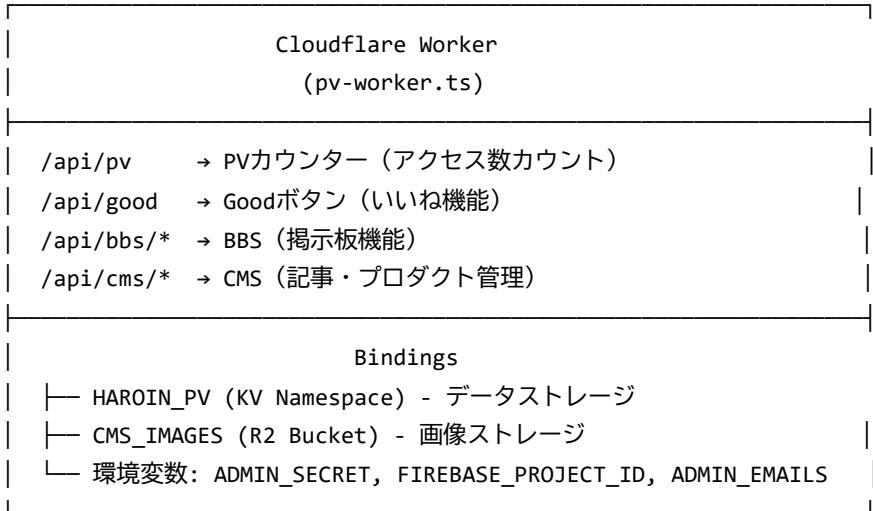
10. 参考リンク

- [Firebase公式: IDトークンの検証](#)
- [JWT.io: JWTのデバッグツール](#)
- [MDN: Web Crypto API](#)
- [RFC 7519: JSON Web Token \(JWT\)](#)

pv-worker.ts 詳細解説

`src/pv-worker.ts` はCloudflare Workerとして動作するバックエンドAPIです。PVカウンター、いいねボタン、BBS、CMSの4つの機能を1つのWorkerで提供しています。

1. 概要とアーキテクチャ



2. 型定義

2.1 環境変数の型

```
type Env = {  
    HAROIN_PV: KVNamespace; // Cloudflare KV (データ保存)  
    CMS_IMAGES?: R2Bucket; // R2バケット (画像アップロード用)  
    ALLOWED_ORIGIN?: string; // CORSで許可するオリジン  
    ADMIN_SECRET?: string; // 管理者用シークレットキー  
    FIREBASE_PROJECT_ID?: string; // Firebase プロジェクトID  
    ADMIN_EMAILS?: string; // 管理者メールアドレス (カンマ区切り)  
    R2_PUBLIC_URL?: string; // R2の公開URL  
}
```

2.2 BBSの型

```
type Thread = {
  id: string          // スレッドID（例: "m1abc2def"）
  title: string        // スレッドタイトル
  createdAt: string   // 作成日時（ISO 8601形式）
  createdBy: string   // 作成者名
  postCount: number   // 投稿数
  lastPostAt: string  // 最終投稿日時
}

type Post = {
  id: number           // 投稿番号（1から連番）
  name: string          // 投稿者名
  date: string          // 投稿日時（2ch形式: "2025/01/15(水) 12:34:56.78"）
  userId: string         // ユーザーID（IPベースで生成、9文字）
  content: string        // 投稿内容
}
```

2.3 CMSの型

```
type CMSPostMeta = {
  slug: string          // URL識別子
  title: string          // タイトル
  summary: string         // 概要
  createdAt: string       // 作成日時
  updatedAt: string       // 更新日時
  tags: string[]          // タグ配列
  status: 'draft' | 'published' // 公開状態
}

type CMSPost = CMSPostMeta & {
  markdown: string      // Markdownソース
  html: string           // レンダリング済みHTML
}

type CMSProductMeta = {
  slug: string          // URL識別子
  name: string          // プロダクト名
  description: string    // 説明
  language: string        // 使用言語
  tags: string[]          // タグ配列
  url: string            // GitHubリポジトリURL
  demo?: string           // デモURL（オプション）
  createdAt: string
  updatedAt: string
}
```

3. ユーティリティ関数

3.1 CORS処理

```
function buildCorsHeaders(origin: string) {
  return {
    'access-control-allow-origin': origin,
    'access-control-allow-methods': 'GET, POST, PUT, PATCH, DELETE, OPTIONS',
    'access-control-allow-headers': 'Content-Type, X-Admin-Secret, Authorization',
  }
}
```

すべてのレスポンスにCORSヘッダーを付与。 X-Admin-Secret と Authorization ヘッダーを許可。

3.2 ユーザーID生成

```
function generateUserId(ip: string): string {
  const hash = simpleHash(ip)
  return hash.slice(0, 9) // 9文字のID
}

function simpleHash(str: string): string {
  // 簡易ハッシュ関数
  // IPアドレスから決定論的なIDを生成
}
```

IPアドレスをハッシュ化して匿名のユーザーIDを生成。同じIPからは常に同じIDになる。

3.3 日時フォーマット（2ch形式）

```
function formatDate(): string {
  // UTC → JST (UTC+9) に変換
  // 出力例: "2025/01/15(水) 12:34:56.78"
}
```

日本時間で2ch風の日時フォーマットを生成。

4. PVカウンター API

エンドポイント: POST /api/pv

```
async function handlePv(req: Request, env: Env, corsHeaders: Promise<Response>
```

処理フロー:

1. IPアドレスからレート制限キーを生成

2. 60秒以内に同一IPからのリクエストがあればカウントせず現在値を返却
3. レート制限に引っかかるなければカウントを+1
4. 新しい合計値を返却

KVキー:

- `rl:{ip}` - レート制限フラグ (TTL: 60秒)
- `total` - 合計PV数

レスポンス例:

```
{ "total": 12345 }
```

5. Goodボタン API

エンドポイント: POST /api/good

```
async function handleGood(req: Request, env: Env, corsHeaders): Promise<Response>
```

リクエストボディ:

```
{
  "slug": "my-article",
  "action": "vote" | "unvote" | "get"
}
```

処理フロー:

1. `action: "get"` → 現在のいいね数と投票済みかを返却
2. `action: "vote"` → いいね数+1、IP記録
3. `action: "unvote"` → いいね数-1、IP記録削除

KVキー:

- `good:{slug}:count` - いいね総数
- `good:{slug}:ip:{ip}` - 投票済みフラグ

レスポンス例:

```
{ "total": 42, "voted": true }
```

6. BBS API

6.1 スレッド一覧取得

```
GET /api/bbs/threads
```

最終投稿日時順にソートされたスレッド一覧を返却。

6.2 スレッド作成

```
POST /api/bbs/threads  
Body: { "title": "スレタイ", "name": "名前", "content": "本文" }
```

レート制限: 同一IPから60秒に1回まで。最大100スレッドまで。

6.3 スレッド取得

```
GET /api/bbs/threads/:id
```

スレッドメタ情報と全投稿を返却。

6.4 投稿追加

```
POST /api/bbs/threads/:id/posts  
Body: { "name": "名前", "content": "本文" }
```

レート制限: 同一IPから60秒に1回まで。1スレッド最大1000投稿まで。

6.5 スレッド削除（管理者のみ）

```
DELETE /api/bbs/threads/:id  
Headers: Authorization: Bearer {Firebase IDトークン}
```

6.6 投稿削除（管理者のみ）

```
DELETE /api/bbs/threads/:id/posts/:postId  
Headers: Authorization: Bearer {Firebase IDトークン}
```

投稿内容を「この投稿は削除されました」に置換。

7. CMS API

7.1 記事一覧

```
GET /api/cms/posts
```

公開済み（status: "published"）の記事のみ返却。作成日時降順。

7.2 下書き一覧（管理者のみ）

```
GET /api/cms/posts/drafts  
Headers: Authorization: Bearer {Firebase IDトークン}
```

7.3 記事詳細

```
GET /api/cms/posts/:slug
```

下書き記事は管理者のみ閲覧可能。

7.4 記事作成（管理者のみ）

```
POST /api/cms/posts  
Headers: Authorization: Bearer {Firebase IDトークン}  
Body: {  
  "slug": "my-article",  
  "title": "タイトル",  
  "summary": "概要",  
  "markdown": "# 本文...",  
  "html": "<h1>本文...</h1>",  
  "tags": ["React", "TypeScript"],  
  "status": "draft" | "published"  
}
```

7.5 記事更新（管理者のみ）

```
PUT /api/cms/posts/:slug  
Headers: Authorization: Bearer {Firebase IDトークン}
```

7.6 記事ステータス変更（管理者のみ）

```
PATCH /api/cms/posts/:slug/status  
Headers: Authorization: Bearer {Firebase IDトークン}  
Body: { "status": "draft" | "published" }
```

7.7 記事削除（管理者のみ）

```
DELETE /api/cms/posts/:slug  
Headers: Authorization: Bearer {Firebase IDトークン}
```

7.8 プロダクトAPI

記事と同様の構造で /api/cms/products/* エンドポイントを提供。

7.9 画像アップロード（管理者のみ）

```
POST /api/cms/upload  
Headers:  
  Authorization: Bearer {Firebase IDトークン}  
  Content-Type: image/png (または image/jpeg など)  
Body: バイナリ画像データ
```

R2バケットに保存し、公開URLを返却。

```
{  
  "key": "1705312345678-abc123.png",  
  "url": "https://images.haroin57.com/1705312345678-abc123.png"  
}
```

8. 管理者認証

8.1 認証方式

2つの認証方式をOR条件でサポート：

```

async function checkAdminAuth(req: Request, env: Env): Promise<boolean> {
  // 方法1: Firebase認証（ブラウザからのリクエスト用）
  const authHeader = req.headers.get('Authorization')
  if (authHeader?.startsWith('Bearer ')) {
    const token = authHeader.slice(7)
    const payload = await verifyFirebaseToken(token, env)
    if (payload?.email && adminEmails.includes(payload.email.toLowerCase())) {
      return true
    }
  }
}

// 方法2: ADMIN_SECRET (CLIツールからのリクエスト用)
const secret = req.headers.get('X-Admin-Secret')
if (env.ADMIN_SECRET && secret === env.ADMIN_SECRET) {
  return true
}

return false
}

```

方式	ヘッダー	用途
Firebase認証	Authorization: Bearer {token}	ブラウザからのリクエスト
ADMIN_SECRET	X-Admin-Secret: {secret}	CLIツール・スクリプト

8.2 Firebase IDトークン検証

verifyFirebaseToken 関数で以下を検証：

- JWTの構造解析:** Header.Payload.Signatureに分解
- アルゴリズム検証:** alg === "RS256" のみ許可
- 時刻検証:**
 - exp > now (有効期限チェック)
 - iat <= now (発行時刻チェック)
 - auth_time <= now (認証時刻チェック)
- 発行者検証:** iss === "https://securetoken.google.com/{project_id}"
- オーディエンス検証:** aud === project_id
- メール検証:** email_verified === true
- RS256署名検証:** Googleの公開鍵で署名を検証

8.3 公開鍵の取得とキャッシュ

```
const GOOGLE_CERTS_URL = 'https://www.googleapis.com/robot/v1/metadata/x509/securetoken@system.gserviceaccount.com'

async function getGooglePublicKeys(): Promise<Record<string, CryptoKey>> {
    // キャッシュが有効な場合はそれを返す
    if (publicKeyCache && publicKeyCache.expiresAt > now) {
        return publicKeyCache.keys
    }

    // Googleから証明書を取得
    const response = await fetch(GOOGLE_CERTS_URL)
    const certs = await response.json()

    // X.509証明書からCryptoKeyに変換
    for (const [kid, pem] of Object.entries(certs)) {
        keys[kid] = await importPublicKeyFromCert(pem)
    }

    // Cache-Controlヘッダーに基づいてキャッシュ
    publicKeyCache = { keys, expiresAt: now + maxAge * 1000 }
    return keys
}
```

- Googleの公開鍵は定期的にローテーションされる
- JWTヘッダーの kid で対応する公開鍵を特定
- Cache-Control ヘッダーに基づいてキャッシュ

8.4 X.509証明書の解析

```
function extractSpkiFromCertificate(certBytes: Uint8Array): ArrayBuffer {
    // ASN.1 DER形式のX.509証明書をパース
    // tbsCertificate内のsubjectPublicKeyInfo（7番目のフィールド）を抽出
    // Web Crypto APIで使えるSPKI形式に変換
}
```

GoogleはPEM形式のX.509証明書を提供するため、ASN.1パースが必要。

9. CORSとセキュリティ

9.1 Origin/Refererチェック

```
const allowedOrigin = env.ALLOWED_ORIGIN || 'https://haroin57.com'

const isAllowed =
  origin === allowedOrigin ||
  (origin === '' && referer.startsWith(allowedOrigin)) ||
  (origin === '' && referer === '')

if (!isAllowed) {
  return new Response('forbidden', { status: 403 })
}
```

許可されたオリジン以外からのリクエストを拒否。

9.2 Preflight対応

```
if (req.method === 'OPTIONS') {
  return new Response(null, { status: 204, headers: corsHeaders })
}
```

10. KVデータ構造

```
HAROIN_PV KV Namespace
├── total                      # 合計PV数
├── rl:{ip}                    # レート制限フラグ (TTL: 60秒)
├── good:{slug}:count          # いいね数
├── good:{slug}:ip:{ip}         # 投票済みフラグ
├── bbs:threads:list           # スレッド一覧 (JSON配列)
├── bbs:thread:{id}:meta       # スレッドメタ情報
├── bbs:thread:{id}:posts      # スレッド投稿一覧
├── bbs:rl:thread:{ip}          # スレッド作成レート制限
├── bbs:rl:post:{ip}            # 投稿レート制限
├── cms:posts:list              # 記事メタ一覧
├── cms:post:{slug}             # 記事データ
├── cms:products:list           # プロダクトメタ一覧
└── cms:product:{slug}          # プロダクトデータ
```

11. デプロイ

```
# wrangler.pv.jsonc を使用してデプロイ
wrangler deploy --config wrangler.pv.jsonc
```

wrangler.pv.jsonc

```
{  
  "name": "haroin-pv",  
  "main": "src/pv-worker.ts",  
  "compatibility_date": "2025-12-02",  
  "kv_namespaces": [  
    { "binding": "HAROIN_PV", "id": "..." }  
  ],  
  "r2_buckets": [  
    { "binding": "CMS_IMAGES", "bucket_name": "haroin-cms-images" }  
  ],  
  "vars": {  
    "R2_PUBLIC_URL": "https://images.haroin57.com"  
  }  
}
```

Secrets (機密情報)

```
wrangler secret put ADMIN_SECRET  
wrangler secret put FIREBASE_PROJECT_ID  
wrangler secret put ADMIN_EMAILS
```

ローカル記事のデプロイ

ローカル環境で作成したMarkdownファイルをCloudflare KVのCMSにデプロイするための機能について解説します。

1. 概要

Web上のエディタだけでなく、VSCodeなどのローカルエディタで記事を書き、それをコマンド一つでCMSにデプロイできます。これにより：

- 使い慣れたエディタで執筆可能
- Gitでの記事バージョン管理
- 複数記事の一括デプロイ
- CI/CDパイプラインとの連携

2. Markdownファイルの形式

記事は `content/posts/` ディレクトリにMarkdownファイルとして配置します。

2.1 ディレクトリ構造

```
content/
└── posts/
    ├── my-first-article.md
    ├── react-tutorial.md
    └── typescript-tips.md
```

ファイル名（拡張子を除く）が記事のslug（URL識別子）になります。

2.2 frontmatter（メタデータ）

各Markdownファイルの先頭にYAML形式でメタデータを記述します：

```
---
title: "記事タイトル"
summary: "記事の概要（一覧ページで表示される説明文）"
date: "2025-01-15"
tags:
  - React
  - TypeScript
  - Tutorial
status: published # オプション: published または draft
---
```

本文をここに書きます…

フィールド	必須	説明
title	○	記事のタイトル
summary	△	記事の概要（省略可）
date	△	作成日（YYYY-MM-DD形式）
tags	△	タグの配列
status	△	published（公開）または draft（下書き）

3. デプロイスクリプトの使い方

3.1 Firebase IDトークンの取得

デプロイには管理者認証が必要です。ブラウザの開発者ツールで以下を実行してトークンを取得します：

```
// Firebaseにログイン済みの状態で実行
const user = firebase.auth().currentUser;
const token = await user.getIdToken();
console.log(token);
```

または、Webサイトにログイン後、コンソールで：

```
// AdminAuthContextを使用している場合
const { idToken } = useAdminAuth();
console.log(idToken);
```

3.2 基本的な使い方

```
# 環境変数にトークンを設定
export FIREBASE_ID_TOKEN="取得したトークン"

# content/posts/ 配下のすべての記事をデプロイ
npx tsx scripts/deploy-posts.ts

# 特定のファイルのみデプロイ
npx tsx scripts/deploy-posts.ts --file content/posts/my-article.md

# 下書きとしてデプロイ
npx tsx scripts/deploy-posts.ts --draft

# ドライラン（実際にはデプロイせず確認のみ）
npx tsx scripts/deploy-posts.ts --dry-run
```

3.3 コマンドラインオプション

オプション	説明	例
--file <path>	指定したファイルのみをデプロイ	--file content/posts/article.md
--draft	下書きとしてデプロイ (frontmatterのstatusを上書き)	--draft
--dry-run	実際にはデプロイせず、処理内容を表示	--dry-run
--endpoint <url>	CMS APIのエンドポイント	--endpoint http://localhost:8787/api/cms
--token <token>	Firebase ID トークン (環境変数の代わりに指定)	--token eyJhbGci...

3.4 環境変数

変数名	説明
FIREBASE_ID_TOKEN	Firebase ID トークン（認証用）
CMS_ENDPOINT	CMS API のエンドポイント（デフォルト: https://haroin57.com/api/cms ）

4. 使用例

4.1 新しい記事を書いて公開する

```
# 1. 記事ファイルを作成
cat > content/posts/new-article.md << 'EOF'
---
title: "新しい記事"
summary: "この記事では..."
date: "2025-01-15"
tags:
- Tech
---
## はじめに

記事の本文をここに書きます。
EOF
```

```
# 2. デプロイ
export FIREBASE_ID_TOKEN="your-token"
npx tsx scripts/deploy-posts.ts --file content/posts/new-article.md
```

4.2 下書きとして保存し、後で公開する

```
# 下書きとしてデプロイ
npx tsx scripts/deploy-posts.ts --file content/posts/draft-article.md --draft

# 後で公開する場合は、frontmatterのstatusをpublishedに変更して再デプロイ
# または、Webの管理画面からステータスを変更
```

4.3 すべての記事を一括更新

```
# 既存の記事も含めてすべてデプロイ
npx tsx scripts/deploy-posts.ts

# まずドライランで確認
npx tsx scripts/deploy-posts.ts --dry-run
```

5. Markdownの記法

デプロイスクリプトは、Webエディタと同じMarkdown変換処理を使用しています。

5.1 サポートされる構文

- **GitHub Flavored Markdown (GFM)**: テーブル、取り消し線、タスクリストなど
- **目次の自動生成**: ## 目次 という見出しの後に自動的に目次が挿入されます
- **コードハイライト**: シンタックスハイライト付きコードブロック
- **数式 (KaTeX)**: $\$inline\$$ や $\$display\$$ 形式の数式
- **Mermaidダイアグラム**: `mermaid` コードブロック
- **アドモニション**: `[!NOTE]`、`[!WARNING]`、`[!CALLOUT]`

5.2 アドモニションの例

- > `[!NOTE]`
これはメモです。重要な情報を強調したいときに使用します。
- > `[!WARNING]`
これは警告です。注意が必要な内容を示します。
- > `[!CALLOUT]` タイトル
これはコールアウトです。特に目立たせたい内容に使用します。

6. トラブルシューティング

6.1 認証エラー

Error: unauthorized

原因: トークンが無効または期限切れ

対処法:

1. 新しいトークンを取得する
2. トークンが正しく設定されているか確認
3. 管理者メールリストに登録されているか確認

6.2 記事が見つからない

Error: post not found

原因: 更新しようとした記事が存在しない

対処法: スクリプトは自動的に新規作成か更新かを判定するため、通常は発生しません。APIエンドポイントを確認してください。

6.3 frontmatterのパースエラー

原因: YAMLの形式が正しくない

対処法:

- インデントが正しいか確認（スペース2つ）
- 特殊文字を含むタイトルは引用符で囲む
- 日付は "YYYY-MM-DD" 形式で記述

7. CI/CDとの連携

GitHub Actionsでの自動デプロイ例：

```
name: Deploy Posts

on:
  push:
    paths:
      - 'content/posts/**'
    branches:
      - main

jobs:
  deploy:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4

      - uses: actions/setup-node@v4
        with:
          node-version: '20'

      - run: npm ci

      - name: Deploy posts
        env:
          FIREBASE_ID_TOKEN: ${{ secrets.FIREBASE_ID_TOKEN }}
        run: npx tsx scripts/deploy-posts.ts
```

注意: CI/CDでのトークン管理には注意が必要です。長期間有効なサービスアカウントトークンの使用を検討してください。