補足資料：ページ初期化パターン

基本(1分) - ページ初期化とは

概要: Webページが読み込まれた直後に一度だけ実行したい処理（データ復元・初期表示・イベントリスナー登録など）をまとめて行うための重要なパターンです。これにより、ユーザーがページを操作する前に必要な準備が整います。

主な目的:

- **データ復元:** `localStorage` などから保存されたデータを読み込み、アプリケーションの状態を復元します。
- **初期表示:** 取得したデータや定義済みの情報に基づいて、ページの初期コンテンツを描画・設定します。
- **イベントリスナー登録:** ボタンクリックやフォーム送信など、ユーザーインタラクションに応答するためのイベントリスナーを設定します。

基本構文（`DOMContentLoaded`）:

```
// HTMLドキュメントの解析が完了し、DOMツリーが構築された時点で実行されます。
// 画像やスタイルシートなどの外部リソースの読み込み完了を待つ必要がない場合に適しています。
document.addEventListener('DOMContentLoaded', function() {
  // ここに初期化処理を記述します
  console.log('DOMの準備ができました。初期化処理を開始します。');
  // 例: ToDoリストのタスクを読み込んで表示する
  // loadTasks();
  // renderTasks();
  // setupEventListeners();
});
```

シンプルな例:

```
document.addEventListener('DOMContentLoaded', function() {
  // ToDoリストのタスクをlocalStorageから読み込んで表示する関数（仮）
  function loadAndDisplayTasks() {
    const tasks = JSON.parse(localStorage.getItem('todoTasks')) || [];
    const taskListElement = document.getElementById('taskList'); // 事前にHTMLに <ul id="taskList"></ul> がある想定
    if (taskListElement) {
      taskListElement.innerHTML = ''; // リストをクリア
      tasks.forEach(task => {
        const listItem = document.createElement('li');
        listItem.textContent = task.text;
        taskListElement.appendChild(listItem);
      });
```

```
    }
    console.log('タスクを読み込み、表示しました。');
  }

  loadAndDisplayTasks();
  // 他の初期化処理（例: 進捗表示の更新）
  // updateProgress();
});
```

詳細(5分) - 初期化処理の仕組みと主要イベント

DOMContentLoaded vs window.onload

Webページの初期化処理を実装する際、主に2つのイベントが利用されます。

1. **DOMContentLoaded イベント:**
   - **発火タイミング:** HTMLドキュメントの解析が完了し、DOM (Document Object Model) ツリーが完全に構築された直後に発火します。
   - **特徴:** スタイルシート、画像、サブフレームなどの外部リソースの読み込み完了を待ちません。そのため、DOM操作を伴う初期化処理をより早く開始できます。
   - **ユースケース:** DOM要素へのアクセス、イベントリスナーの登録、`localStorage` からのデータ読み込みと表示など、スクリプトがDOM構造のみに依存する場合に最適です。

```
document.addEventListener('DOMContentLoaded', function() {
  // DOMは準備完了、画像などはまだ読み込み中かもしれない
  const mainTitle = document.getElementById('main-title');
  if (mainTitle) {
    mainTitle.textContent = 'ようこそ！';
  }
  // イベントリスナーの設定
  // setupFormValidation();
});
```

2. **window.onload イベント:**
   - **発火タイミング:** HTMLドキュメントとすべての外部リソース（画像、スタイルシート、スクリプト、サブフレームなど）の読み込みが完了した後に発火します。
   - **特徴:** ページ全体の準備が整った状態で処理を開始できますが、`DOMContentLoaded` よりも発火が遅くなる傾向があります。

- **ユースケース:** 画像のサイズを取得したり、外部スクリプトに依存する処理を行ったりする場合など、すべてのリソースが利用可能である必要がある場合に用います。

```
window.onload = function() {
  // ページ上の全てのコンテンツ（画像も含む）が読み込み完了
  const heroImage = document.getElementById('hero-image');
  if (heroImage) {
    console.log(`ヒーロー画像のサイズ: ${heroImage.offsetWidth}x${heroImage.offsetHeight}`);
  }
  // initializeThirdPartyAnalytics();
};
```

どちらを使うべきか？ 一般的には、`DOMContentLoaded` **を優先的に使用する**ことが推奨されます。DOM操作はHTMLの解析完了直後から可能であり、ユーザーに早くインタラクティブなページを提供できるためです。画像や外部スクリプトの読み込み完了を待つ必要がない処理は、`DOMContentLoaded` で行うのが効率的です。

よくある間違いと注意点

- **グローバルスコープでの直接的なDOM操作:**

```
// ❌ 間違った例: スクリプトが<head>内や<body>の早い段階で読み込まれると、
// 対象のDOM要素がまだ解析・構築されていない可能性がある。
// const myButton = document.getElementById('myButton'); // myButton がまだ存在しないかも
// myButton.addEventListener('click', myFunction); // エラーになる可能性
```

✅ **正しい例:** `DOMContentLoaded` や `window.onload` を使用して、DOMの準備ができてから操作を行います。

```
document.addEventListener('DOMContentLoaded', function() {
  const myButton = document.getElementById('myButton');
  if (myButton) {
    myButton.addEventListener('click', myFunction);
  }
});
function myFunction() { /* ... */ }
```

- **初期化関数の肥大化:**
  - 初期化処理が長大になると、可読性やメンテナンス性が低下します。関連する処理ごとに小さな関数に分割し、初期化関数内からそれらを呼び出すようにしましょう。

```
document.addEventListener('DOMContentLoaded', function() {
  initializeUserData();
```

```
  renderPageLayout();
  setupGlobalEventHandlers();
  checkInitialPermissions();
});

function initializeUserData() { /* ... */ }
function renderPageLayout() { /* ... */ }
// ... 他の関数
```

深掘り（5分） - 非同期処理とエラーハンドリング

非同期処理を伴う初期化

現代のWebアプリケーションでは、初期化時に外部APIからデータを取得するなど、非同期処理が必要になるケースが多くあります。`async/await` や `Promise` を使うことで、非同期処理を伴う初期化を分かりやすく記述できます。

例: `async/await` を使用したデータ取得と表示

```
document.addEventListener('DOMContentLoaded', async function() {
  console.log('初期化開始: ユーザーデータを取得します...');
  try {
    const userData = await fetchUserData(); // 非同期でユーザーデータを取得する関数
    displayUserProfile(userData); // 取得したデータでプロフィールを表示
    console.log('ユーザーデータの取得と表示が完了しました。');

    const settings = await fetchAppSettings(); // 非同期でアプリ設定を取得
    applySettings(settings); // 設定を適用
    console.log('アプリ設定の取得と適用が完了しました。');

  } catch (error) {
    console.error('初期化処理中にエラーが発生しました:', error);
    displayErrorMessage('データの読み込みに失敗しました。ページを再読み込みしてください。');
    // 必要に応じて、ユーザーにエラーメッセージを表示する処理
  }
});

// 外部APIからユーザーデータを取得する非同期関数の例（実際にはfetch APIなどを使用）
async function fetchUserData() {
  // return fetch('/api/user').then(response => response.json()); // 本来の処理
  return new Promise(resolve => setTimeout(() => resolve({ name: '山田太郎', theme: 'dark' }), 1000)); // ダミー
}

// 外部APIからアプリ設定を取得する非同期関数の例
```

```
async function fetchAppSettings() {
  // return fetch('/api/settings').then(response => response.json()); // 本来の処理
  return new Promise(resolve => setTimeout(() => resolve({ notifications: true, language: 'ja' }), 500)); // ダミー
}

function displayUserProfile(data) {
  const profileElement = document.getElementById('userProfile');
  if (profileElement) profileElement.textContent = `ようこそ、${data.name}さん`;
  console.log('プロフィール表示:', data);
}

function applySettings(settings) {
  // 設定を適用するロジック（例: ダークモードの切り替えなど）
  if (settings.theme === 'dark') {
    document.body.classList.add('dark-theme');
  }
  console.log('設定適用:', settings);
}

function displayErrorMessage(message) {
  const errorElement = document.getElementById('errorMessage'); // HTMLに <div id="errorMessage"></div> がある想定
  if(errorElement) errorElement.textContent = message;
}
```

この例では、`fetchUserData` と `fetchAppSettings` が非同期に実行され、それぞれのデータ取得後にUI更新処理が行われます。

初期化処理中のエラーハンドリング

初期化処理はアプリケーションの起動時に不可欠なため、エラーが発生した場合の対処が重要です。`try...catch` ブロックを使用して、予期せぬエラーを捕捉し、適切に処理します。

エラーハンドリングのポイント:

- **ユーザーへのフィードバック:** エラーが発生したことをユーザーに伝え、可能な対処法（例: 再読み込み）を提示します。
- **ログ記録:** 開発者向けにエラーの詳細をコンソールに出力したり、エラー監視サービスに送信したりします。
- **フォールバック処理:** 重要なデータの読み込みに失敗した場合、デフォルト値を使用する、機能を制限するなどのフォールバック処理を検討します。

```
document.addEventListener('DOMContentLoaded', function() {
  try {
    // localStorageから設定を読み込む
    const settingsString = localStorage.getItem('userSettings');
    if (settingsString) {
      const settings = JSON.parse(settingsString); // JSON.parseは失敗する可能性あり
      applyTheme(settings.theme);
    } else {
```

```
    // 設定がない場合はデフォルトテーマを適用
    applyTheme('light');
  }
  initializeCriticalFeature(); // この関数もエラーを投げる可能性
} catch (error) {
  console.error('初期化中にエラーが発生しました:', error.name, error.message);
  // ユーザーに汎用的なエラーメッセージを表示
  const errorDisplay = document.getElementById('initialization-error');
  if (errorDisplay) {
    errorDisplay.textContent = 'ページの読み込み中に問題が発生しました。一部機能が利用できない可能性があります。';
  }
  // 開発者向けに詳細なエラー情報をログに残す
  // logErrorToServer(error);

  // 補足: try-catchの詳細については、別資料「try-catch詳解」を参照してください。
}
});

function applyTheme(themeName) {
  console.log(`テーマを ${themeName} に設定します。`);
  // document.body.className = `${themeName}-theme`; // 実際のテーマ適用処理
}

function initializeCriticalFeature() {
  // 重要な機能の初期化ロジック
  // 例: 必須データのチェック
  // if (!checkRequiredData()) {
  //   throw new Error('必須データが不足しているため、重要機能を初期化できません。');
  // }
  console.log('重要機能の初期化が完了しました。');
}
```

エラーハンドリングを適切に行うことで、アプリケーションの堅牢性が向上します。詳細は **補足資料: try-catch詳解** を参照してください。

実践応用(5分) - 様々な初期化シナリオ

パターン1: `localStorage` からのデータ復元と画面描画

ToDoリストアプリケーションなどで、以前保存したタスクを `localStorage` から読み込み、ページに表示するケースです。

```
document.addEventListener('DOMContentLoaded', function() {
  const taskList = document.getElementById('todo-list'); // <ul id="todo-list"></ul>
  const newTaskInput = document.getElementById('new-task-input'); // <input id="new-task-input">
```

```
const addTaskButton = document.getElementById('add-task-button'); // <button id="add-task-button">

let tasks = [];

// 1. localStorageからタスクを読み込む関数
function loadTasksFromStorage() {
  try {
    const storedTasks = localStorage.getItem('todoAppTasks');
    if (storedTasks) {
      tasks = JSON.parse(storedTasks);
    }
  } catch (e) {
    console.error('タスクの読み込みに失敗しました:', e);
    tasks = []; // エラー時は空のリストで初期化
  }
}

// 2. タスクリストを画面に描画する関数
function renderTasks() {
  if (!taskList) return;
  taskList.innerHTML = ''; // 既存のリストアイテムをクリア
  tasks.forEach((task, index) => {
    const listItem = document.createElement('li');
    listItem.textContent = task.text;
    // (省略) 削除ボタンなどの追加
    taskList.appendChild(listItem);
  });
}

// 3. タスクをlocalStorageに保存する関数
function saveTasksToStorage() {
  try {
    localStorage.setItem('todoAppTasks', JSON.stringify(tasks));
  } catch (e) {
    console.error('タスクの保存に失敗しました:', e);
    // 保存失敗時のユーザー通知などを検討
  }
}

// 4. 新しいタスクを追加する処理
function addTask() {
  if (!newTaskInput || !newTaskInput.value.trim()) return;
  tasks.push({ text: newTaskInput.value.trim(), completed: false });
  newTaskInput.value = ''; // 入力欄をクリア
  renderTasks();
  saveTasksToStorage();
}

// 初期化処理の実行
loadTasksFromStorage();
renderTasks();

// イベントリスナーの設定
if (addTaskButton) {
  addTaskButton.addEventListener('click', addTask);
```

```
  }
  // （省略）Enterキーでのタスク追加など
});
```

このパターンは、**補足資料: localStorage API詳解** や **補足資料: JSON操作詳解** と密接に関連します。

パターン2: ユーザー設定の読み込みと適用

ユーザーが以前に設定したテーマ（例: ダークモード）や表示設定を `localStorage` から読み込み、ページの表示に反映させるケースです。

```
document.addEventListener('DOMContentLoaded', function() {
  const themeToggleButton = document.getElementById('theme-toggle'); // <button id="theme-toggle">

  let currentTheme = 'light'; // デフォルトテーマ

  // 1. localStorageからテーマ設定を読み込む
  function loadThemeSetting() {
    try {
      const storedTheme = localStorage.getItem('userPreferredTheme');
      if (storedTheme && (storedTheme === 'light' || storedTheme === 'dark')) {
        currentTheme = storedTheme;
      }
    } catch (e) {
      console.error('テーマ設定の読み込みエラー:', e);
      // エラー時はデフォルトテーマのまま
    }
  }

  // 2. テーマをページに適用する
  function applyTheme() {
    document.body.classList.remove('light-theme', 'dark-theme');
    document.body.classList.add(`${currentTheme}-theme`);
    if (themeToggleButton) {
      themeToggleButton.textContent = currentTheme === 'light' ? 'ダークモードに切り替え' : 'ライトモードに切り替え';
    }
    console.log(`テーマを ${currentTheme} に設定しました。`);
  }

  // 3. テーマ設定をlocalStorageに保存する
  function saveThemeSetting() {
    try {
      localStorage.setItem('userPreferredTheme', currentTheme);
    } catch (e) {
      console.error('テーマ設定の保存エラー:', e);
    }
  }

  // 4. テーマ切り替え処理
  function toggleTheme() {
```

```
    currentTheme = currentTheme === 'light' ? 'dark' : 'light';
    applyTheme();
    saveThemeSetting();
  }

  // 初期化処理
  loadThemeSetting();
  applyTheme();

  // イベントリスナー
  if (themeToggleButton) {
    themeToggleButton.addEventListener('click', toggleTheme);
  }
});
```

パターン3: SPA (Single Page Application) における初期化

React, Vue, AngularなどのSPAフレームワークでは、独自のライフサイクルメソッドや初期化の仕組みが提供されています。

- **React:** `useEffect` フック（空の依存配列 `[]` を指定）やクラスコンポーネントの `componentDidMount`。
- **Vue:** `mounted` ライフサイルフック。
- **Angular:** `ngOnInit` ライフサイルフック。

これらのフレームワークを使用する場合は、フレームワークの作法に従って初期化処理を実装します。 `DOMContentLoaded` を直接使うことは少なくなりますが、基本的な考え方は共通です。

技術選択の判断基準まとめ

- **DOM操作のみで完結する処理:** `DOMContentLoaded` を使用します。これにより、ユーザーはより早くインタラティブなページを利用できます。
- **画像や外部ファイルの読み込み完了が必要な処理:** `window.onload` を使用します。ただし、発火が遅れる可能性があるため、本当に必要か検討しましょう。
- **非同期処理を含む初期化:** `async/await` や `Promise` を活用し、`DOMContentLoaded` 内で実行します。エラーハンドリングも忘れずに行います。
- **SPAフレームワーク利用時:** 各フレームワークが提供するライフサイクルメソッドや初期化の仕組みを利用します。
- **エラーハンドリング:** すべての初期化処理において `try...catch` を適切に配置し、堅牢性を高めます。

関連する補足資料

- **補足資料: localStorage API詳解** - データ復元・保存の具体的な方法。
- **補足資料: JSON操作詳解** - `localStorage` でデータを扱う際の `JSON.parse` / `JSON.stringify` の使い方と注意点。
- **補足資料: try-catch詳解** - エラーハンドリングの基本と実践。
- **補足資料: DOM_イベント処理.md** - `addEventListener` を含む、DOMイベントのより詳細な扱い方。
- **補足資料: フロントエンド_状態管理.md** - 初期化時にアプリケーションの状態をどのように設定・管理するかの参考。
- **（必要に応じて）補足資料: 非同期処理入門.md** - `Promise` や `async/await` の基礎。