補足資料: DOM イベント処理

基本(1分) - DOMイベント処理とは

概要: DOMイベント処理とは、Webページ上でユーザーの操作（クリック、入力、キーボード操作など）やブラウザの状態変化を検知し、それに応じてJavaScriptで処理を実行する仕組みです。インタラクティブなWebアプリケーションを作成するために不可欠な技術です。

なぜDOMイベント処理が重要か:

- **ユーザーインタラクション**: ボタンクリック、フォーム入力など、ユーザーの操作に反応できます。
- **動的なUI**: ページの内容をユーザーの操作に応じてリアルタイムで変更できます。
- **データの同期**: ユーザーの入力をすぐにlocalStorageなどに保存できます。
- **UX向上**: フィードバックの即座な提供により、使いやすいアプリケーションを実現できます。

基本構文:

```
// イベントリスナーの登録
element.addEventListener('eventType', function(event) {
  // イベント発生時の処理
});

// または矢印関数で
element.addEventListener('eventType', (event) => {
  // イベント発生時の処理
});
```

主要なイベントタイプ:

- `click`: 要素がクリックされた時
- `input`: 入力要素の値が変更された時（リアルタイム）
- `change`: 入力要素の値が変更され、フォーカスが外れた時
- `submit`: フォームが送信された時
- `keydown` / `keyup`: キーが押された時/離された時

シンプルな例:

```
// HTML: <button id="addButton">タスクを追加</button>
const addButton = document.getElementById('addButton');

addButton.addEventListener('click', function(event) {
  console.log('ボタンがクリックされました！');
  // ToDoタスクの追加処理をここに書く
});

// 入力フィールドの変更を検知
// HTML: <input id="taskInput" type="text" placeholder="新しいタスク">
const taskInput = document.getElementById('taskInput');

taskInput.addEventListener('input', function(event) {
  console.log('入力内容:', event.target.value);
  // リアルタイムで入力内容を保存する処理をここに書く
});
```

詳細(3分) - DOMイベント処理の仕組みと詳細

イベントリスナーの詳細

1. addEventListener() の構文と オプション

```
element.addEventListener(eventType, handler, options);

// options の例
element.addEventListener('click', handler, {
  once: true,        // 一度だけ実行
  passive: true,     // preventDefault() を呼ばない（パフォーマンス向上）
  capture: false     // キャプチャフェーズで実行するかどうか
});
```

2. イベントハンドラーの定義方法

```
const button = document.getElementById('myButton');

// 方法1: 無名関数
button.addEventListener('click', function(event) {
  console.log('クリックされました');
});

// 方法2: 矢印関数
```

```
button.addEventListener('click', (event) => {
  console.log('クリックされました');
});

// 方法3: 名前付き関数（推奨：再利用・削除が可能）
function handleButtonClick(event) {
  console.log('クリックされました');
}
button.addEventListener('click', handleButtonClick);
```

イベントオブジェクトの活用

イベントハンドラーの第一引数として渡されるイベントオブジェクトには、イベントに関する詳細情報が含まれています：

```
function handleClick(event) {
  console.log('イベントタイプ:', event.type);          // 'click'
  console.log('対象要素:', event.target);              // クリックされた要素
  console.log('現在の要素:', event.currentTarget);     // イベントリスナーが設定された要素
  console.log('マウス座標:', event.clientX, event.clientY); // クリック位置
  console.log('押されたキー:', event.key);             // キーイベントの場合

  // デフォルト動作の阻止
  event.preventDefault();

  // イベントの伝播を止める
  event.stopPropagation();
}

// フォーム送信イベントの例
const form = document.getElementById('todoForm');
form.addEventListener('submit', function(event) {
  event.preventDefault(); // ページリロードを防ぐ
  console.log('フォームデータ:', new FormData(event.target));
});
```

よく使用されるイベントタイプの詳細

1. マウスイベント

```
const element = document.getElementById('taskItem');

element.addEventListener('click', (e) => console.log('クリック'));
element.addEventListener('dblclick', (e) => console.log('ダブルクリック'));
element.addEventListener('mousedown', (e) => console.log('マウスボタン押下'));
element.addEventListener('mouseup', (e) => console.log('マウスボタン離す'));
```

```
element.addEventListener('mouseover', (e) => console.log('マウス進入'));
element.addEventListener('mouseout', (e) => console.log('マウス退出'));
```

2. キーボードイベント

```
const input = document.getElementById('taskInput');

input.addEventListener('keydown', function(event) {
  console.log('押されたキー:', event.key);

  if (event.key === 'Enter') {
    console.log('Enterキーが押されました');
    // タスク追加処理を実行
  }

  if (event.ctrlKey && event.key === 's') {
    event.preventDefault(); // Ctrl+Sのデフォルト動作（保存ダイアログ）を阻止
    console.log('Ctrl+Sが押されました');
    // 手動保存処理を実行
  }
});
```

3. フォームイベント

```
const input = document.getElementById('taskInput');

// リアルタイムで入力を検知
input.addEventListener('input', function(event) {
  console.log('現在の入力値:', event.target.value);
  // 自動保存や入力値検証を実行
});

// フォーカスが外れた時に検知
input.addEventListener('change', function(event) {
  console.log('最終的な入力値:', event.target.value);
  // 入力完了時の処理を実行
});

// フォーカスイベント
input.addEventListener('focus', (e) => console.log('フォーカス取得'));
input.addEventListener('blur', (e) => console.log('フォーカス喪失'));
```

イベントの伝播（バブリング・キャプチャリング）

DOMでは、イベントが発生すると3つのフェーズで処理されます：

```
// HTML構造の例
// <div id="container">
//   <button id="button">クリック</button>
// </div>

const container = document.getElementById('container');
const button = document.getElementById('button');

// キャプチャフェーズ（親→子）
container.addEventListener('click', () => {
  console.log('親要素（キャプチャ）');
}, true); // 第3引数をtrueにするとキャプチャフェーズで実行

// バブリングフェーズ（子→親）デフォルト
button.addEventListener('click', () => {
  console.log('ボタン（ターゲット）');
});

container.addEventListener('click', () => {
  console.log('親要素（バブリング）');
});

// ボタンをクリックした時の出力順序:
// 1. 親要素（キャプチャ）
// 2. ボタン（ターゲット）
// 3. 親要素（バブリング）
```

イベントリスナーの削除

メモリリークを防ぐため、不要になったイベントリスナーは削除する必要があります：

```
function handleClick(event) {
  console.log('クリックされました');
}

const button = document.getElementById('myButton');

// イベントリスナーを登録
button.addEventListener('click', handleClick);

// イベントリスナーを削除（同じ関数参照を使用）
button.removeEventListener('click', handleClick);

// 無名関数は削除できない例（悪い例）
button.addEventListener('click', function() {
  console.log('削除できません');
});

// AbortController を使用した削除（モダンな方法）
const controller = new AbortController();
```

```
button.addEventListener('click', handleClick, {
  signal: controller.signal
});

// 一括削除
controller.abort(); // このコントローラーに関連するすべてのイベントリスナーを削除
```

よくある間違いと注意点

❌ 間違った例1: 無名関数の多用

```
// 削除やデバッグが困難
button.addEventListener('click', function() {
  // 処理
});
```

❌ 間違った例2: イベントリスナーの重複登録

```
// 同じイベントに複数回登録してしまう
for (let i = 0; i < 5; i++) {
  button.addEventListener('click', handleClick); // 5回登録される
}
```

❌ 間違った例3: thisの誤用

```
const taskManager = {
  count: 0,
  handleClick: function(event) {
    this.count++; // thisはここではbuttonを指すことがある
  }
};

button.addEventListener('click', taskManager.handleClick); // 予期しない動作
```

✅ 正しい例1: 名前付き関数の使用

```
function handleButtonClick(event) {
  // 処理
}
button.addEventListener('click', handleButtonClick);
```

✅ 正しい例2: 重複チェック

```
// 既にイベントリスナーが登録されているかチェック
if (!button.hasAttribute('data-listener-added')) {
  button.addEventListener('click', handleClick);
  button.setAttribute('data-listener-added', 'true');
}
```

✅ 正しい例3: bind() またはアロー関数でthisを固定

```
const taskManager = {
  count: 0,
  handleClick: function(event) {
    this.count++;
  }
};

// bind() を使用
button.addEventListener('click', taskManager.handleClick.bind(taskManager));

// またはアロー関数
button.addEventListener('click', (event) => taskManager.handleClick(event));
```

深掘り(コラム) - DOMイベント処理と関連技術

関連技術マップ（優先度順）

🔥 重要度：高 - Event Delegation（イベントデリゲーション）

関連性: 動的に追加される要素（ToDoリストアイテムなど）のイベント処理を効率的に管理するために必須です。

```
// 悪い例：各タスクにイベントリスナーを個別登録
function addTaskItem(taskText) {
  const taskItem = document.createElement('div');
  taskItem.innerHTML = `
    <span>${taskText}</span>
    <button class="delete-btn">削除</button>
  `;
```

```
  // 毎回新しいイベントリスナーを登録（非効率）
  const deleteBtn = taskItem.querySelector('.delete-btn');
  deleteBtn.addEventListener('click', (e) => {
    taskItem.remove();
  });

  taskList.appendChild(taskItem);
}

// 良い例：親要素でイベントデリゲーション
const taskList = document.getElementById('taskList');

taskList.addEventListener('click', function(event) {
  // 削除ボタンがクリックされた場合
  if (event.target.classList.contains('delete-btn')) {
    const taskItem = event.target.closest('.task-item');
    taskItem.remove();
  }

  // チェックボックスがクリックされた場合
  if (event.target.classList.contains('task-checkbox')) {
    const taskItem = event.target.closest('.task-item');
    taskItem.classList.toggle('completed');
  }
});

// 新しいタスクの追加は、イベントリスナーの再登録が不要
function addTaskItem(taskText) {
  const taskItem = document.createElement('div');
  taskItem.className = 'task-item';
  taskItem.innerHTML = `
    <input type="checkbox" class="task-checkbox">
    <span>${taskText}</span>
    <button class="delete-btn">削除</button>
  `;
  taskList.appendChild(taskItem);
}
```

🔥 重要度：高 - カスタムイベント

関連性: アプリケーション独自のイベントを作成し、コンポーネント間の通信を実現するために重要です。

```
// カスタムイベントの作成と発火
class TaskManager {
  addTask(taskText) {
    const task = { id: Date.now(), text: taskText, completed: false };
    this.tasks.push(task);

    // カスタムイベントを発火
    const event = new CustomEvent('taskAdded', {
```

```
      detail: { task: task },
      bubbles: true
    });
    document.dispatchEvent(event);
  }

  completeTask(taskId) {
    const task = this.tasks.find(t => t.id === taskId);
    if (task) {
      task.completed = true;

      // タスク完了イベントを発火
      const event = new CustomEvent('taskCompleted', {
        detail: { task: task }
      });
      document.dispatchEvent(event);
    }
  }
}

// カスタムイベントのリスナー
document.addEventListener('taskAdded', function(event) {
  console.log('新しいタスクが追加されました:', event.detail.task);
  // localStorage に保存
  saveTasksToStorage();
  // UI の更新
  updateTaskCount();
});

document.addEventListener('taskCompleted', function(event) {
  console.log('タスクが完了しました:', event.detail.task);
  // 完了エフェクトの表示
  showCompletionEffect();
});

// 使用例
const taskManager = new TaskManager();
taskManager.addTask('牛乳を買う');
```

🔶 重要度：中 – パフォーマンス最適化（スロットリング・デバウンシング）

関連性: リアルタイム検索や自動保存などで、イベントの実行頻度を制御するために重要です。

```
// デバウンシング：連続するイベントの最後のみ実行
function debounce(func, wait) {
  let timeout;
  return function executedFunction(...args) {
    const later = () => {
      clearTimeout(timeout);
      func(...args);
    };
```

```
    clearTimeout(timeout);
    timeout = setTimeout(later, wait);
  };
}

// スロットリング：指定した間隔で実行
function throttle(func, limit) {
  let inThrottle;
  return function(...args) {
    if (!inThrottle) {
      func.apply(this, args);
      inThrottle = true;
      setTimeout(() => inThrottle = false, limit);
    }
  }
}

// 使用例：リアルタイム検索
const searchInput = document.getElementById('searchInput');

// デバウンシング：ユーザーが入力を止めてから300ms後に検索実行
const debouncedSearch = debounce(function(query) {
  console.log('検索実行:', query);
  performSearch(query);
}, 300);

searchInput.addEventListener('input', function(event) {
  debouncedSearch(event.target.value);
});

// スロットリング：スクロール位置の監視（最大100msに1回）
const throttledScroll = throttle(function() {
  console.log('スクロール位置:', window.scrollY);
  // スクロール位置に応じた処理
}, 100);

window.addEventListener('scroll', throttledScroll);
```

🔶 重要度：中 - Promise ベースのイベント処理

関連性: 非同期処理とイベント処理を組み合わせて、よりモダンなコードを書くために有用です。

```
// Promise ベースのイベント待機
function waitForEvent(element, eventType) {
  return new Promise((resolve) => {
    element.addEventListener(eventType, resolve, { once: true });
  });
}

// 使用例：ユーザーのクリックを待つ
async function waitForUserConfirmation() {
```

```
  const confirmButton = document.getElementById('confirmButton');
  const cancelButton = document.getElementById('cancelButton');

  return new Promise((resolve) => {
    const handleClick = (event) => {
      if (event.target === confirmButton) {
        resolve(true);
      } else if (event.target === cancelButton) {
        resolve(false);
      }

      // イベントリスナーを削除
      confirmButton.removeEventListener('click', handleClick);
      cancelButton.removeEventListener('click', handleClick);
    };

    confirmButton.addEventListener('click', handleClick);
    cancelButton.addEventListener('click', handleClick);
  });
}

// 使用例
async function deleteTask() {
  const confirmed = await waitForUserConfirmation();
  if (confirmed) {
    console.log('タスクを削除します');
  } else {
    console.log('削除をキャンセルしました');
  }
}
```

🔹 重要度：補足 - Intersection Observer API との連携

関連性: 要素の可視性に基づいたイベント処理を効率的に実現するために有用です。

```
// Intersection Observer を使用した可視化イベント
const observer = new IntersectionObserver((entries) => {
  entries.forEach(entry => {
    if (entry.isIntersecting) {
      // 要素が見える時のイベント
      const customEvent = new CustomEvent('elementVisible', {
        detail: { element: entry.target }
      });
      entry.target.dispatchEvent(customEvent);
    } else {
      // 要素が見えなくなった時のイベント
      const customEvent = new CustomEvent('elementHidden', {
        detail: { element: entry.target }
      });
      entry.target.dispatchEvent(customEvent);
    }
```

```
  });
});

// タスクアイテムの可視性を監視
document.querySelectorAll('.task-item').forEach(taskItem => {
  observer.observe(taskItem);

  taskItem.addEventListener('elementVisible', () => {
    console.log('タスクが表示されました');
    // 遅延読み込みやアニメーション処理
  });

  taskItem.addEventListener('elementHidden', () => {
    console.log('タスクが非表示になりました');
    // クリーンアップ処理
  });
});
```

技術の全体像

これらの技術は、以下のように組み合わせて使用されます：

1. **基本的なイベント処理**: ユーザーインタラクションの基盤
2. **Event Delegation**: 効率的な動的要素の管理
3. **カスタムイベント**: コンポーネント間通信
4. **パフォーマンス最適化**: 高頻度イベントの制御
5. **Promise ベース処理**: 非同期イベント処理
6. **Observer API**: 効率的な状態監視

実践応用 - DOMイベント処理の総合活用

パターン1: ToDoリストアプリケーション（包括的実装）

使用場面: DOMイベント処理を中心とした完全なToDoアプリケーション

```
class TodoApp {
  constructor() {
    this.tasks = [];
    this.currentFilter = 'all'; // 'all', 'active', 'completed'
    this.init();
  }

  init() {
```

```
  this.loadTasksFromStorage();
  this.setupEventListeners();
  this.render();
}

setupEventListeners() {
  const form = document.getElementById('todoForm');
  const taskInput = document.getElementById('taskInput');
  const filterButtons = document.querySelectorAll('.filter-btn');
  const clearCompletedBtn = document.getElementById('clearCompleted');
  const selectAllBtn = document.getElementById('selectAll');

  // フォーム送信イベント
  form.addEventListener('submit', (event) => {
    event.preventDefault();
    this.addTask(taskInput.value.trim());
    taskInput.value = '';
  });

  // リアルタイム入力保存（デバウンシング）
  const debouncedSave = this.debounce(() => {
    sessionStorage.setItem('taskDraft', taskInput.value);
  }, 300);

  taskInput.addEventListener('input', debouncedSave);

  // 下書きの復元
  const draft = sessionStorage.getItem('taskDraft');
  if (draft) {
    taskInput.value = draft;
  }

  // Enterキーでのタスク追加
  taskInput.addEventListener('keydown', (event) => {
    if (event.key === 'Enter' && !event.shiftKey) {
      event.preventDefault();
      form.dispatchEvent(new Event('submit'));
    }
  });

  // タスクリストでのイベントデリゲーション
  const taskList = document.getElementById('taskList');
  taskList.addEventListener('click', this.handleTaskListClick.bind(this));
  taskList.addEventListener('change', this.handleTaskListChange.bind(this));

  // フィルターボタン
  filterButtons.forEach(btn => {
    btn.addEventListener('click', (event) => {
      this.setFilter(event.target.dataset.filter);
      this.updateFilterButtons();
      this.render();
    });
  });

  // 完了済みタスクの一括削除
```

```
  clearCompletedBtn.addEventListener('click', () => {
    this.clearCompletedTasks();
  });

  // 全選択/全解除
  selectAllBtn.addEventListener('click', () => {
    this.toggleAllTasks();
  });

  // カスタムイベントのリスナー
  document.addEventListener('taskUpdated', () => {
    this.saveTasksToStorage();
    this.render();
  });

  // ページ離脱時の確認
  window.addEventListener('beforeunload', (event) => {
    const incompleteTasks = this.tasks.filter(task => !task.completed);
    if (incompleteTasks.length > 0) {
      event.returnValue = '未完了のタスクがあります。本当にページを離れますか？';
    }
  });

  // ショートカットキー
  document.addEventListener('keydown', (event) => {
    if (event.ctrlKey || event.metaKey) {
      switch (event.key) {
        case 's':
          event.preventDefault();
          this.saveTasksToStorage();
          this.showMessage('タスクを保存しました');
          break;
        case 'a':
          event.preventDefault();
          this.toggleAllTasks();
          break;
      }
    }
  });
}

handleTaskListClick(event) {
  const taskItem = event.target.closest('.task-item');
  if (!taskItem) return;

  const taskId = parseInt(taskItem.dataset.taskId);

  if (event.target.classList.contains('delete-btn')) {
    this.deleteTask(taskId);
  } else if (event.target.classList.contains('edit-btn')) {
    this.editTask(taskId, taskItem);
  } else if (event.target.classList.contains('task-text')) {
    // タスクテキストをダブルクリックで編集
    if (event.detail === 2) { // ダブルクリック
      this.editTask(taskId, taskItem);
```

```
    }
  }
}

handleTaskListChange(event) {
  if (event.target.classList.contains('task-checkbox')) {
    const taskItem = event.target.closest('.task-item');
    const taskId = parseInt(taskItem.dataset.taskId);
    this.toggleTask(taskId);
  }
}

addTask(text) {
  if (!text) return;

  const task = {
    id: Date.now(),
    text: text,
    completed: false,
    createdAt: new Date().toISOString()
  };

  this.tasks.push(task);
  sessionStorage.removeItem('taskDraft'); // 下書きを削除

  // カスタムイベントを発火
  const event = new CustomEvent('taskAdded', {
    detail: { task: task }
  });
  document.dispatchEvent(event);

  this.dispatchTaskUpdated();
}

toggleTask(taskId) {
  const task = this.tasks.find(t => t.id === taskId);
  if (task) {
    task.completed = !task.completed;
    task.updatedAt = new Date().toISOString();

    // 完了アニメーション
    if (task.completed) {
      this.showCompletionEffect(taskId);
    }

    this.dispatchTaskUpdated();
  }
}

deleteTask(taskId) {
  const taskIndex = this.tasks.findIndex(t => t.id === taskId);
  if (taskIndex > -1) {
    const deletedTask = this.tasks[taskIndex];
    this.tasks.splice(taskIndex, 1);
```

```
    // 削除の取り消し機能
    this.showUndoMessage('タスクを削除しました', () => {
      this.tasks.splice(taskIndex, 0, deletedTask);
      this.dispatchTaskUpdated();
    });

    this.dispatchTaskUpdated();
  }
}

editTask(taskId, taskElement) {
  const task = this.tasks.find(t => t.id === taskId);
  if (!task) return;

  const textElement = taskElement.querySelector('.task-text');
  const originalText = task.text;

  // インライン編集用の入力フィールドを作成
  const input = document.createElement('input');
  input.type = 'text';
  input.value = originalText;
  input.className = 'edit-input';

  // テキスト要素を入力フィールドに置き換え
  textElement.style.display = 'none';
  textElement.parentNode.insertBefore(input, textElement.nextSibling);
  input.focus();
  input.select();

  const finishEditing = (save = true) => {
    if (save && input.value.trim() !== '') {
      task.text = input.value.trim();
      task.updatedAt = new Date().toISOString();
      textElement.textContent = task.text;
      this.dispatchTaskUpdated();
    }

    textElement.style.display = '';
    input.remove();
  };

  // Enterキーで保存、Escapeキーでキャンセル
  input.addEventListener('keydown', (event) => {
    if (event.key === 'Enter') {
      finishEditing(true);
    } else if (event.key === 'Escape') {
      finishEditing(false);
    }
  });

  // フォーカス離脱で保存
  input.addEventListener('blur', () => {
    finishEditing(true);
  });
}
```

```
  showCompletionEffect(taskId) {
    const taskElement = document.querySelector(`[data-task-id="${taskId}"]`);
    if (taskElement) {
      taskElement.classList.add('completion-effect');
      setTimeout(() => {
        taskElement.classList.remove('completion-effect');
      }, 1000);
    }
  }

  showUndoMessage(message, undoCallback) {
    const notification = document.createElement('div');
    notification.className = 'undo-notification';
    notification.innerHTML = `
      ${message}
      <button class="undo-btn">取り消し</button>
    `;

    document.body.appendChild(notification);

    const undoBtn = notification.querySelector('.undo-btn');
    undoBtn.addEventListener('click', () => {
      undoCallback();
      notification.remove();
    });

    // 5秒後に自動削除
    setTimeout(() => {
      if (notification.parentNode) {
        notification.remove();
      }
    }, 5000);
  }

  debounce(func, wait) {
    let timeout;
    return function executedFunction(...args) {
      const later = () => {
        clearTimeout(timeout);
        func(...args);
      };
      clearTimeout(timeout);
      timeout = setTimeout(later, wait);
    };
  }

  dispatchTaskUpdated() {
    const event = new CustomEvent('taskUpdated');
    document.dispatchEvent(event);
  }

  // その他のメソッド（render, saveTasksToStorage等）は省略...
}
```

```
// CSS（完了効果のアニメーション）
const style = document.createElement('style');
style.textContent = `
  .completion-effect {
    animation: completionPulse 1s ease-in-out;
  }

  @keyframes completionPulse {
    0% { transform: scale(1); }
    50% { transform: scale(1.05); background-color: #d4edda; }
    100% { transform: scale(1); }
  }

  .undo-notification {
    position: fixed;
    bottom: 20px;
    left: 50%;
    transform: translateX(-50%);
    background: #333;
    color: white;
    padding: 10px 20px;
    border-radius: 5px;
    z-index: 1000;
  }

  .undo-btn {
    background: #007bff;
    color: white;
    border: none;
    padding: 5px 10px;
    margin-left: 10px;
    border-radius: 3px;
    cursor: pointer;
  }
`;
document.head.appendChild(style);

// アプリケーション初期化
const todoApp = new TodoApp();
```

パターン2: localStorage + DOMイベント + リアルタイム同期

使用場面: ユーザーの操作をリアルタイムでlocalStorageに同期する

```
class AutoSaveTodoApp {
  constructor() {
    this.storageKey = 'autoSaveTodos';
    this.tasks = this.loadTasks();
    this.setupAutoSave();
  }

  setupAutoSave() {
```

```
    // 全てのイベントに対して自動保存を設定
    const events = ['taskAdded', 'taskUpdated', 'taskDeleted', 'taskToggled'];

    events.forEach(eventType => {
      document.addEventListener(eventType, () => {
        this.saveTasksWithRetry();
      });
    });

    // 定期的な保存（フォールバック）
    setInterval(() => {
      this.saveTasksWithRetry();
    }, 30000); // 30秒ごと
  }

  async saveTasksWithRetry(maxRetries = 3) {
    for (let i = 0; i < maxRetries; i++) {
      try {
        localStorage.setItem(this.storageKey, JSON.stringify(this.tasks));
        console.log('タスクを保存しました');
        return true;
      } catch (error) {
        console.warn(`保存試行 ${i + 1} 失敗:`, error.message);

        if (error.name === 'QuotaExceededError') {
          // ストレージ容量不足の場合、古いデータを削除
          this.cleanupOldData();
        }

        await new Promise(resolve => setTimeout(resolve, 1000)); // 1秒待機
      }
    }

    console.error('保存に失敗しました');
    this.showSaveErrorMessage();
    return false;
  }

  cleanupOldData() {
    // 30日以上前のタスクを削除
    const thirtyDaysAgo = new Date(Date.now() - 30 * 24 * 60 * 60 * 1000);
    const originalCount = this.tasks.length;

    this.tasks = this.tasks.filter(task => {
      const taskDate = new Date(task.createdAt);
      return taskDate > thirtyDaysAgo;
    });

    const removedCount = originalCount - this.tasks.length;
    if (removedCount > 0) {
      console.log(`${removedCount} 件の古いタスクを削除しました`);
    }
  }
}
```

技術選択の判断基準まとめ

状況A（シンプルなインタラクション）の場合: 基本的な `addEventListener` と名前付き関数を使用。

状況B（動的な要素が多い）の場合: Event Delegation を使用し、効率的なイベント管理を行う。

状況C（高頻度のイベント処理）の場合: デバウンシングやスロットリングを適用してパフォーマンスを最適化。

状況D（複雑なアプリケーション）の場合: カスタムイベントとPromiseベースの処理を組み合わせて、保守性の高い構造を構築。

重要な原則:

1. **イベントリスナーの適切な管理（登録・削除）**
2. **パフォーマンスを意識した実装**
3. **ユーザビリティを向上させるリアルタイム処理**
4. **エラーハンドリングとフォールバック処理**

関連する補足資料

- localStorage API詳解: DOMイベントで取得したデータをlocalStorageに保存する方法。
- JSON操作詳解: イベントで取得したオブジェクトデータをJSON形式で保存・読み込みする方法。
- try-catch詳解: DOMイベント処理中のエラーハンドリング方法。
- JavaScript オブジェクト詳解: イベントオブジェクトやカスタムデータの詳細な操作方法。
- フロントエンド 状態管理: DOMイベントを通じたアプリケーション状態の管理方法。

最終更新日: 2024/06/13