localStorageパフォーマンス

> 関連資料: localStorage API詳解 | データ整合性管理 | ブラウザストレージ比較

1. 基本（1分で理解）

localStorageのパフォーマンスとは、データの読み書き速度とメモリ使用量の効率化を指します。localStorageは**同期的な操作**のため、不適切な使用はUIの応答性を低下させます。

パフォーマンスに影響する要因

- **データサイズ**：大きなJSONオブジェクトの変換コスト
- **アクセス頻度**：頻繁な読み書きによるブロッキング
- **JSON処理**：`JSON.stringify/parse`の計算コスト
- **ストレージ容量**：約5MBの制限による制約

```
// ❌ パフォーマンスが悪い例
function addTask(task) {
    const tasks = JSON.parse(localStorage.getItem('tasks') || '[]'); // 毎回全体を読み取り
    tasks.push(task);
    localStorage.setItem('tasks', JSON.stringify(tasks)); // 毎回全体を保存
}

// ✅ パフォーマンスが良い例
class OptimizedTaskManager {
    constructor() {
        this.cache = null; // メモリキャッシュ
        this.isDirty = false; // 変更フラグ
    }

    getTasks() {
        if (!this.cache) {
            this.cache = JSON.parse(localStorage.getItem('tasks') || '[]');
        }
        return this.cache;
    }
```

```
    addTask(task) {
        this.getTasks().push(task);
        this.isDirty = true;
        this.debouncedSave(); // 遅延保存
    }
}
```

2. 詳細（3分で習得）

2.1 パフォーマンス測定

実際の性能を測定してボトルネックを特定します。

```
class PerformanceMonitor {
    constructor() {
        this.metrics = {
            readTime: [],
            writeTime: [],
            parseTime: [],
            stringifyTime: []
        };
    }

    // localStorage読み取り性能の測定
    measureRead(key) {
        const start = performance.now();
        const value = localStorage.getItem(key);
        const readEnd = performance.now();

        const parseStart = performance.now();
        const data = value ? JSON.parse(value) : null;
        const parseEnd = performance.now();

        this.metrics.readTime.push(readEnd - start);
        this.metrics.parseTime.push(parseEnd - parseStart);

        return data;
    }

    // localStorage書き込み性能の測定
    measureWrite(key, data) {
        const stringifyStart = performance.now();
        const value = JSON.stringify(data);
        const stringifyEnd = performance.now();

        const writeStart = performance.now();
```

```
        localStorage.setItem(key, value);
        const writeEnd = performance.now();

        this.metrics.stringifyTime.push(stringifyEnd - stringifyStart);
        this.metrics.writeTime.push(writeEnd - writeStart);
    }

    // パフォーマンスレポートの生成
    generateReport() {
        const report = {};

        for (const [metric, times] of Object.entries(this.metrics)) {
            if (times.length > 0) {
                report[metric] = {
                    average: times.reduce((a, b) => a + b, 0) / times.length,
                    min: Math.min(...times),
                    max: Math.max(...times),
                    count: times.length
                };
            }
        }

        return report;
    }

    // パフォーマンスアラートの設定
    checkThresholds() {
        const thresholds = {
            readTime: 10,      // 10ms以上は警告
            writeTime: 50,     // 50ms以上は警告
            parseTime: 20,     // 20ms以上は警告
            stringifyTime: 30  // 30ms以上は警告
        };

        const warnings = [];

        for (const [metric, threshold] of Object.entries(thresholds)) {
            const times = this.metrics[metric];
            if (times.length > 0) {
                const avg = times.reduce((a, b) => a + b, 0) / times.length;
                if (avg > threshold) {
                    warnings.push(`${metric}の平均時間が${avg.toFixed(2)}msで閾値${threshold}msを超えています`);
                }
            }
        }

        return warnings;
    }
}
```

2.2 メモリキャッシュ戦略

頻繁なlocalStorageアクセスを避けるためのキャッシュ実装：

```
class CachedTaskStorage {
    constructor() {
        this.cache = new Map();
        this.isDirty = new Set();
        this.lastSave = new Map();

        // 定期的な自動保存
        setInterval(() => this.flushDirtyData(), 5000);

        // ページ離脱時の保存
        window.addEventListener('beforeunload', () => this.flushAll());
    }

    // データの取得（キャッシュ優先）
    getData(key) {
        if (this.cache.has(key)) {
            return this.cache.get(key);
        }

        // キャッシュミス：localStorageから読み取り
        try {
            const stored = localStorage.getItem(key);
            const data = stored ? JSON.parse(stored) : null;
            this.cache.set(key, data);
            return data;
        } catch (error) {
            console.error(`データ読み取りエラー (${key}):`, error);
            return null;
        }
    }

    // データの設定（キャッシュに保存、遅延書き込み）
    setData(key, data) {
        this.cache.set(key, data);
        this.isDirty.add(key);
        this.lastSave.set(key, Date.now());

        // 即座に保存が必要な場合のオプション
        if (this.shouldSaveImmediately(key, data)) {
            this.saveToStorage(key);
        }
    }

    // 即座に保存すべきかの判定
    shouldSaveImmediately(key, data) {
        // 重要なデータの場合
        if (key.includes('important') || key.includes('critical')) {
            return true;
        }

        // データサイズが小さい場合
        if (JSON.stringify(data).length < 1000) {
            return true;
```

```
        }

        return false;
    }

    // 変更されたデータのみを保存
    flushDirtyData() {
        for (const key of this.isDirty) {
            this.saveToStorage(key);
        }
    }

    // 全データの強制保存
    flushAll() {
        for (const [key] of this.cache) {
            this.saveToStorage(key);
        }
    }

    // localStorageへの実際の保存
    saveToStorage(key) {
        try {
            const data = this.cache.get(key);
            if (data !== null) {
                localStorage.setItem(key, JSON.stringify(data));
            }
            this.isDirty.delete(key);
        } catch (error) {
            console.error(`データ保存エラー (${key}):`, error);
            this.handleStorageError(key, error);
        }
    }

    // ストレージエラーの処理
    handleStorageError(key, error) {
        if (error.name === 'QuotaExceededError') {
            console.warn('localStorageの容量が不足しています');
            this.cleanupOldData();
        }
    }

    // 古いデータのクリーンアップ
    cleanupOldData() {
        const now = Date.now();
        const maxAge = 7 * 24 * 60 * 60 * 1000; // 7日

        for (const [key] of this.cache) {
            const lastAccess = this.lastSave.get(key) || 0;
            if (now - lastAccess > maxAge) {
                this.removeData(key);
            }
        }
    }

    removeData(key) {
```

```
        this.cache.delete(key);
        this.isDirty.delete(key);
        this.lastSave.delete(key);
        localStorage.removeItem(key);
    }
}
```

2.3 データ圧縮とページング

大量データの効率的な処理 手法：

```
class CompressedTaskStorage {
    constructor() {
        this.pageSize = 100; // 1ページあたりのタスク数
        this.compressionThreshold = 1000; // 圧縮を開始するサイズ（文字数）
    }

    // データの圧縮保存
    saveCompressed(key, data) {
        const jsonString = JSON.stringify(data);

        if (jsonString.length > this.compressionThreshold) {
            // LZ圧縮（簡易版）
            const compressed = this.simpleCompress(jsonString);
            localStorage.setItem(key, compressed);
            localStorage.setItem(`${key}_compressed`, 'true');
        } else {
            localStorage.setItem(key, jsonString);
            localStorage.removeItem(`${key}_compressed`);
        }
    }

    // 圧縮データの読み取り
    loadCompressed(key) {
        const isCompressed = localStorage.getItem(`${key}_compressed`) === 'true';
        const data = localStorage.getItem(key);

        if (!data) return null;

        try {
            if (isCompressed) {
                const decompressed = this.simpleDecompress(data);
                return JSON.parse(decompressed);
            } else {
                return JSON.parse(data);
            }
        } catch (error) {
            console.error('データ復元エラー:', error);
            return null;
        }
    }
```

```
// 簡易LZ圧縮
simpleCompress(str) {
    const dict = {};
    let data = (str + "").split("");
    let out = [];
    let currChar;
    let phrase = data[0];
    let code = 256;

    for (let i = 1; i < data.length; i++) {
        currChar = data[i];
        if (dict[phrase + currChar] != null) {
            phrase += currChar;
        } else {
            out.push(phrase.length > 1 ? dict[phrase] : phrase.charCodeAt(0));
            dict[phrase + currChar] = code;
            code++;
            phrase = currChar;
        }
    }
    out.push(phrase.length > 1 ? dict[phrase] : phrase.charCodeAt(0));

    return btoa(JSON.stringify(out));
}

// 簡易LZ展開
simpleDecompress(str) {
    const dict = {};
    let data = JSON.parse(atob(str));
    let currChar = String.fromCharCode(data[0]);
    let oldPhrase = currChar;
    let out = [currChar];
    let code = 256;
    let phrase;

    for (let i = 1; i < data.length; i++) {
        let currCode = data[i];
        if (currCode < 256) {
            phrase = String.fromCharCode(data[i]);
        } else {
            phrase = dict[currCode] ? dict[currCode] : (oldPhrase + currChar);
        }
        out.push(phrase);
        currChar = phrase.charAt(0);
        dict[code] = oldPhrase + currChar;
        code++;
        oldPhrase = phrase;
    }

    return out.join("");
}

// ページング機能
savePagedTasks(tasks) {
    const totalPages = Math.ceil(tasks.length / this.pageSize);
```

```
        for (let page = 0; page < totalPages; page++) {
            const start = page * this.pageSize;
            const end = start + this.pageSize;
            const pageData = tasks.slice(start, end);

            this.saveCompressed(`tasks_page_${page}`, pageData);
        }

        // メタデータの保存
        localStorage.setItem('tasks_meta', JSON.stringify({
            totalTasks: tasks.length,
            totalPages: totalPages,
            pageSize: this.pageSize,
            lastUpdated: new Date().toISOString()
        }));
    }

    // ページ単位での読み込み
    loadTaskPage(pageNumber) {
        return this.loadCompressed(`tasks_page_${pageNumber}`) || [];
    }

    // 全タスクの遅延読み込み
    async loadAllTasksLazy() {
        const meta = JSON.parse(localStorage.getItem('tasks_meta') || '{}');
        if (!meta.totalPages) return [];

        const allTasks = [];

        for (let page = 0; page < meta.totalPages; page++) {
            const pageData = this.loadTaskPage(page);
            allTasks.push(...pageData);

            // 大量データ読み込み時のUI応答性確保
            if (page % 5 === 0) {
                await new Promise(resolve => setTimeout(resolve, 0));
            }
        }

        return allTasks;
    }
}
```

2.4 遅延保存（Debouncing）とバッチ処理

無駄な書き込みを減らす最適化手法：

```
class DebouncedTaskManager {
    constructor() {
        this.pendingChanges = new Map();
        this.saveTimers = new Map();
```

```
        this.batchQueue = [];
        this.isProcessingBatch = false;

        this.defaultDelay = 300; // デフォルト遅延時間（ミリ秒）
        this.batchSize = 10; // バッチサイズ
        this.maxBatchDelay = 1000; // 最大バッチ遅延
    }

    // 遅延保存の実行
    debouncedSave(key, data, delay = this.defaultDelay) {
        // 以前のタイマーをクリア
        if (this.saveTimers.has(key)) {
            clearTimeout(this.saveTimers.get(key));
        }

        // 変更を保留
        this.pendingChanges.set(key, data);

        // 新しいタイマーを設定
        const timer = setTimeout(() => {
            this.executeSave(key);
        }, delay);

        this.saveTimers.set(key, timer);
    }

    // 実際の保存処理
    executeSave(key) {
        if (this.pendingChanges.has(key)) {
            const data = this.pendingChanges.get(key);

            try {
                localStorage.setItem(key, JSON.stringify(data));
                this.pendingChanges.delete(key);
                this.saveTimers.delete(key);

                console.log(`データ保存完了: ${key}`);
            } catch (error) {
                console.error(`保存エラー: ${key}`, error);
                this.handleSaveError(key, data, error);
            }
        }
    }

    // バッチ処理による一括保存
    addToBatch(operation) {
        this.batchQueue.push({
            ...operation,
            timestamp: Date.now()
        });

        // バッチサイズに達した場合、または最大遅延時間が経過した場合
        if (this.batchQueue.length >= this.batchSize || this.shouldProcessImmediately()) {
            this.processBatch();
        } else if (!this.isProcessingBatch) {
```

```
        // バッチ処理の遅延実行
        setTimeout(() => {
            if (this.batchQueue.length > 0) {
                this.processBatch();
            }
        }, this.maxBatchDelay);
    }
}

shouldProcessImmediately() {
    if (this.batchQueue.length === 0) return false;

    const oldestOperation = this.batchQueue[0];
    const age = Date.now() - oldestOperation.timestamp;

    return age > this.maxBatchDelay;
}

async processBatch() {
    if (this.isProcessingBatch || this.batchQueue.length === 0) return;

    this.isProcessingBatch = true;
    const batch = [...this.batchQueue];
    this.batchQueue = [];

    try {
        // バッチ処理の開始通知
        this.onBatchStart(batch.length);

        for (const operation of batch) {
            await this.processOperation(operation);

            // 処理間の小休止（UI応答性確保）
            await new Promise(resolve => setTimeout(resolve, 1));
        }

        this.onBatchComplete(batch.length);

    } catch (error) {
        console.error('バッチ処理エラー:', error);
        this.onBatchError(error, batch);
    } finally {
        this.isProcessingBatch = false;
    }
}

async processOperation(operation) {
    switch (operation.type) {
        case 'save':
            localStorage.setItem(operation.key, JSON.stringify(operation.data));
            break;
        case 'delete':
            localStorage.removeItem(operation.key);
            break;
        case 'update':
```

```
            const existing = localStorage.getItem(operation.key);
            const data = existing ? JSON.parse(existing) : {};
            Object.assign(data, operation.changes);
            localStorage.setItem(operation.key, JSON.stringify(data));
            break;
    }
}

// 即座に保存（緊急時）
forceSave(key) {
    if (this.pendingChanges.has(key)) {
        this.executeSave(key);
    }
}

// 全ての保留中の変更を保存
flushAll() {
    for (const key of this.pendingChanges.keys()) {
        this.forceSave(key);
    }

    // バッチ処理も強制実行
    if (this.batchQueue.length > 0) {
        this.processBatch();
    }
}

// イベントハンドラー
onBatchStart(count) {
    console.log(`バッチ処理開始: ${count}件の操作`);
}

onBatchComplete(count) {
    console.log(`バッチ処理完了: ${count}件の操作`);
}

onBatchError(error, batch) {
    console.error('バッチ処理でエラーが発生:', error);
    // エラーが発生した操作を再試行キューに追加
    this.batchQueue.unshift(...batch);
}

handleSaveError(key, data, error) {
    if (error.name === 'QuotaExceededError') {
        // 容量不足の場合、古いデータを削除して再試行
        this.cleanupStorage();
        setTimeout(() => this.debouncedSave(key, data), 1000);
    }
}

cleanupStorage() {
    // 古いキャッシュやログデータを削除
    const keysToCheck = [];
    for (let i = 0; i < localStorage.length; i++) {
        const key = localStorage.key(i);
```

```
            if (key.includes('cache_') || key.includes('log_')) {
                keysToCheck.push(key);
            }
        }

        keysToCheck.forEach(key => {
            try {
                localStorage.removeItem(key);
            } catch (e) {
                console.warn(`クリーンアップ失敗: ${key}`);
            }
        });
    }
}
```

3. 深掘りコラム

3.1 localStorage vs IndexedDB vs WebSQL

パフォーマンス比較と適切な選択基準：

```
class StoragePerformanceComparison {
    constructor() {
        this.testData = this.generateTestData(1000); // 1000件のテストタスク
    }

    generateTestData(count) {
        const tasks = [];
        for (let i = 0; i < count; i++) {
            tasks.push({
                id: `task_${i}`,
                text: `タスク ${i} の詳細な説明`.repeat(Math.floor(Math.random() * 10) + 1),
                completed: Math.random() > 0.5,
                priority: ['low', 'normal', 'high'][Math.floor(Math.random() * 3)],
                tags: Array.from({length: Math.floor(Math.random() * 5)}, (_, i) => `tag${i}`),
                createdAt: new Date(Date.now() - Math.random() * 86400000).toISOString(),
                updatedAt: new Date().toISOString()
            });
        }
        return tasks;
    }

    // localStorage性能テスト
    async testLocalStorage() {
        const results = { write: 0, read: 0, size: 0 };

        // 書き込みテスト
```

```
        const writeStart = performance.now();
        localStorage.setItem('test_tasks', JSON.stringify(this.testData));
        results.write = performance.now() - writeStart;

        // サイズ測定
        results.size = localStorage.getItem('test_tasks').length;

        // 読み取りテスト
        const readStart = performance.now();
        JSON.parse(localStorage.getItem('test_tasks'));
        results.read = performance.now() - readStart;

        // クリーンアップ
        localStorage.removeItem('test_tasks');

        return results;
    }

    // IndexedDB性能テスト
    async testIndexedDB() {
        const results = { write: 0, read: 0, size: 0 };

        return new Promise((resolve, reject) => {
            const request = indexedDB.open('PerformanceTest', 1);

            request.onerror = () => reject(request.error);

            request.onsuccess = async () => {
                const db = request.result;

                try {
                    // 書き込みテスト
                    const writeStart = performance.now();
                    const transaction = db.transaction(['tasks'], 'readwrite');
                    const store = transaction.objectStore('tasks');

                    for (const task of this.testData) {
                        store.add(task);
                    }

                    await new Promise(resolve => {
                        transaction.oncomplete = resolve;
                    });
                    results.write = performance.now() - writeStart;

                    // 読み取りテスト
                    const readStart = performance.now();
                    const readTransaction = db.transaction(['tasks'], 'readonly');
                    const readStore = readTransaction.objectStore('tasks');
                    const allTasks = [];

                    await new Promise(resolve => {
                        const cursor = readStore.openCursor();
                        cursor.onsuccess = (event) => {
                            const cursor = event.target.result;
```

```
                        if (cursor) {
                            allTasks.push(cursor.value);
                            cursor.continue();
                        } else {
                            resolve();
                        }
                    };
                });
                results.read = performance.now() - readStart;
                results.size = JSON.stringify(allTasks).length;

                // クリーンアップ
                db.close();
                indexedDB.deleteDatabase('PerformanceTest');

                resolve(results);
            } catch (error) {
                reject(error);
            }
        };

        request.onupgradeneeded = (event) => {
            const db = event.target.result;
            const store = db.createObjectStore('tasks', { keyPath: 'id' });
        };
    });
}

// 包括的な比較実行
async runComparison() {
    console.log('ストレージ性能比較を開始...');

    const localStorageResults = await this.testLocalStorage();
    const indexedDBResults = await this.testIndexedDB();

    const comparison = {
        localStorage: localStorageResults,
        indexedDB: indexedDBResults,
        recommendation: this.generateRecommendation(localStorageResults, indexedDBResults)
    };

    console.table(comparison);
    return comparison;
}

generateRecommendation(localStorage, indexedDB) {
    let recommendation = "";

    if (localStorage.write < indexedDB.write && localStorage.read < indexedDB.read) {
        recommendation = "小規模データではlocalStorageが高速";
    } else if (indexedDB.write < localStorage.write && indexedDB.read < localStorage.read) {
        recommendation = "大規模データではIndexedDBが高速";
    } else {
        recommendation = "データサイズと使用パターンに応じて選択";
    }
```

```
        return recommendation;
    }
}
```

3.2 メモリリーク対策

長時間実行アプリケーションでのメモリ管理：

```
class MemoryOptimizedTaskManager {
    constructor() {
        this.cache = new Map();
        this.cacheStats = {
            hits: 0,
            misses: 0,
            evictions: 0
        };
        this.maxCacheSize = 100; // 最大キャッシュサイズ
        this.ttl = 5 * 60 * 1000; // 5分のTTL

        // 定期的なメモリクリーンアップ
        setInterval(() => this.cleanupMemory(), 60000); // 1分ごと

        // メモリ使用量の監視
        this.setupMemoryMonitoring();
    }

    setupMemoryMonitoring() {
        if ('memory' in performance) {
            setInterval(() => {
                const memory = performance.memory;
                const usage = {
                    used: Math.round(memory.usedJSHeapSize / 1048576), // MB
                    total: Math.round(memory.totalJSHeapSize / 1048576), // MB
                    limit: Math.round(memory.jsHeapSizeLimit / 1048576) // MB
                };

                // メモリ使用量が70%を超えたら警告
                if (usage.used / usage.limit > 0.7) {
                    console.warn('メモリ使用量が高くなっています:', usage);
                    this.aggressiveCleanup();
                }
            }, 10000); // 10秒ごと
        }
    }

    // LRU キャッシュの実装
    get(key) {
        const item = this.cache.get(key);

        if (!item) {
            this.cacheStats.misses++;
```

```
            return null;
        }

        // TTL チェック
        if (Date.now() - item.timestamp > this.ttl) {
            this.cache.delete(key);
            this.cacheStats.misses++;
            return null;
        }

        // LRU: アクセスされたアイテムを最後に移動
        this.cache.delete(key);
        this.cache.set(key, { ...item, timestamp: Date.now() });
        this.cacheStats.hits++;

        return item.data;
    }

    set(key, data) {
        // キャッシュサイズの制限
        if (this.cache.size >= this.maxCacheSize) {
            // 最も古いアイテムを削除
            const firstKey = this.cache.keys().next().value;
            this.cache.delete(firstKey);
            this.cacheStats.evictions++;
        }

        this.cache.set(key, {
            data: data,
            timestamp: Date.now()
        });
    }

    cleanupMemory() {
        const now = Date.now();
        const beforeSize = this.cache.size;

        // 期限切れアイテムの削除
        for (const [key, item] of this.cache.entries()) {
            if (now - item.timestamp > this.ttl) {
                this.cache.delete(key);
            }
        }

        const afterSize = this.cache.size;
        const cleaned = beforeSize - afterSize;

        if (cleaned > 0) {
            console.log(`メモリクリーンアップ: ${cleaned}件のアイテムを削除`);
        }

        // WeakMap によるガベージコレクション最適化
        this.triggerGC();
    }
```

```
    aggressiveCleanup() {
        // 積極的なクリーンアップ
        this.cache.clear();
        this.cacheStats.evictions += this.cache.size;

        // DOM要素の参照削除
        this.cleanupDOMReferences();

        // イベントリスナーの削除
        this.cleanupEventListeners();

        console.log('積極的なメモリクリーンアップを実行しました');
    }

    triggerGC() {
        // ガベージコレクションのヒント
        if (window.gc) {
            window.gc();
        }
    }

    cleanupDOMReferences() {
        // 不要なDOM要素の参照を削除
        const obsoleteElements = document.querySelectorAll('.task-item[data-obsolete="true"]');
        obsoleteElements.forEach(element => element.remove());
    }

    cleanupEventListeners() {
        // 古いイベントリスナーの削除
        // 実装は具体的なUIフレームワークに依存
    }

    getMemoryStats() {
        return {
            cacheSize: this.cache.size,
            cacheStats: this.cacheStats,
            memoryUsage: performance.memory ? {
                used: Math.round(performance.memory.usedJSHeapSize / 1048576),
                total: Math.round(performance.memory.totalJSHeapSize / 1048576)
            } : null
        };
    }
}
```

3.3 プログレッシブ読み込み

大量データの段階的読み込みによる UX 向上：

```
class ProgressiveTaskLoader {
    constructor() {
        this.pageSize = 20;
        this.currentPage = 0;
```

```
        this.isLoading = false;
        this.hasMore = true;
        this.loadedTasks = [];

        this.setupInfiniteScroll();
    }

    async loadInitialTasks() {
        const tasks = await this.loadTaskPage(0);
        this.renderTasks(tasks);
        this.currentPage = 0;
        this.loadedTasks = [...tasks];

        return tasks;
    }

    async loadTaskPage(page) {
        if (this.isLoading) return [];

        this.isLoading = true;
        this.showLoadingIndicator();

        try {
            // ページごとのデータ読み込み
            const startIndex = page * this.pageSize;
            const allTasks = await this.getAllTasksFromStorage();
            const pageTasks = allTasks.slice(startIndex, startIndex + this.pageSize);

            // 読み込み遅延のシミュレーション（実際の処理時間を再現）
            await new Promise(resolve => setTimeout(resolve, 100));

            this.hasMore = startIndex + this.pageSize < allTasks.length;

            return pageTasks;

        } finally {
            this.isLoading = false;
            this.hideLoadingIndicator();
        }
    }

    async loadMoreTasks() {
        if (!this.hasMore || this.isLoading) return;

        const nextPage = this.currentPage + 1;
        const newTasks = await this.loadTaskPage(nextPage);

        if (newTasks.length > 0) {
            this.currentPage = nextPage;
            this.loadedTasks.push(...newTasks);
            this.appendTasks(newTasks);
        }

        return newTasks;
    }
```

```
    setupInfiniteScroll() {
        const container = document.getElementById('taskContainer');
        if (!container) return;

        let ticking = false;

        const checkScroll = () => {
            const { scrollTop, scrollHeight, clientHeight } = container;
            const threshold = 100; // 100px手前でトリガー

            if (scrollHeight - scrollTop - clientHeight < threshold) {
                this.loadMoreTasks();
            }

            ticking = false;
        };

        container.addEventListener('scroll', () => {
            if (!ticking) {
                requestAnimationFrame(checkScroll);
                ticking = true;
            }
        });
    }

    // 仮想スクロールの実装
    setupVirtualScroll() {
        const container = document.getElementById('taskContainer');
        const itemHeight = 60; // 各タスクアイテムの高さ
        const visibleCount = Math.ceil(container.clientHeight / itemHeight) + 5; // バッファ

        let startIndex = 0;
        let endIndex = visibleCount;

        const updateVirtualView = () => {
            const scrollTop = container.scrollTop;
            const newStartIndex = Math.floor(scrollTop / itemHeight);
            const newEndIndex = newStartIndex + visibleCount;

            if (newStartIndex !== startIndex || newEndIndex !== endIndex) {
                startIndex = newStartIndex;
                endIndex = newEndIndex;
                this.renderVirtualTasks(startIndex, endIndex);
            }
        };

        container.addEventListener('scroll', () => {
            requestAnimationFrame(updateVirtualView);
        });
    }

    renderVirtualTasks(startIndex, endIndex) {
        const container = document.getElementById('taskList');
        const visibleTasks = this.loadedTasks.slice(startIndex, endIndex);
```

```
        // コンテナの高さを全体の高さに設定
        const totalHeight = this.loadedTasks.length * 60;
        container.style.height = `${totalHeight}px`;

        // 現在表示する範囲のみをレンダリング
        const visibleContainer = document.getElementById('visibleTasks') ||
                                 document.createElement('div');
        visibleContainer.id = 'visibleTasks';
        visibleContainer.style.transform = `translateY(${startIndex * 60}px)`;

        visibleContainer.innerHTML = '';
        visibleTasks.forEach(task => {
            const element = this.createTaskElement(task);
            visibleContainer.appendChild(element);
        });

        if (!visibleContainer.parentNode) {
            container.appendChild(visibleContainer);
        }
    }

    async getAllTasksFromStorage() {
        // ここでは効率的なデータ読み込みを実装
        const chunks = [];
        let chunkIndex = 0;

        while (true) {
            const chunkKey = `tasks_chunk_${chunkIndex}`;
            const chunk = localStorage.getItem(chunkKey);

            if (!chunk) break;

            try {
                chunks.push(...JSON.parse(chunk));
            } catch (error) {
                console.error(`チャンク ${chunkIndex} の読み込みエラー:`, error);
            }

            chunkIndex++;
        }

        return chunks;
    }

    showLoadingIndicator() {
        const indicator = document.getElementById('loadingIndicator');
        if (indicator) {
            indicator.style.display = 'block';
        }
    }

    hideLoadingIndicator() {
        const indicator = document.getElementById('loadingIndicator');
        if (indicator) {
```

```
            indicator.style.display = 'none';
        }
    }
}
```

4. 実践応用

4.1 高性能Todoアプリの完全実装

```
class HighPerformanceTodoApp {
    constructor() {
        // パフォーマンス最適化コンポーネント
        this.performanceMonitor = new PerformanceMonitor();
        this.cachedStorage = new CachedTaskStorage();
        this.debouncedManager = new DebouncedTaskManager();
        this.memoryManager = new MemoryOptimizedTaskManager();
        this.progressiveLoader = new ProgressiveTaskLoader();

        // 設定
        this.config = {
            autoSaveInterval: 5000,
            maxTasksInMemory: 500,
            compressionThreshold: 1000,
            performanceAlertThreshold: 100
        };

        this.initializeApp();
    }

    async initializeApp() {
        try {
            // パフォーマンス監視の開始
            this.startPerformanceMonitoring();

            // 初期データの読み込み
            await this.loadInitialData();

            // UI のセットアップ
            this.setupEventListeners();
            this.setupAutoSave();

            console.log('高性能Todoアプリが初期化されました');

        } catch (error) {
            console.error('アプリ初期化エラー:', error);
            this.handleInitializationError(error);
        }
```

```
}

startPerformanceMonitoring() {
    // 定期的なパフォーマンスチェック
    setInterval(() => {
        const report = this.performanceMonitor.generateReport();
        const warnings = this.performanceMonitor.checkThresholds();

        if (warnings.length > 0) {
            console.warn('パフォーマンス警告:', warnings);
            this.optimizePerformance();
        }
    }, 30000); // 30秒ごと
}

async loadInitialData() {
    const loadStart = performance.now();

    try {
        // キャッシュから高速読み込み
        let tasks = this.memoryManager.get('allTasks');

        if (!tasks) {
            // キャッシュミス：ストレージから読み込み
            tasks = this.performanceMonitor.measureRead('tasks');
            this.memoryManager.set('allTasks', tasks);
        }

        // プログレッシブ読み込みでUI更新
        await this.progressiveLoader.loadInitialTasks();

        const loadTime = performance.now() - loadStart;
        console.log(`初期データ読み込み完了: ${loadTime.toFixed(2)}ms`);

    } catch (error) {
        console.error('データ読み込みエラー:', error);
        throw error;
    }
}

// 高性能なタスク追加
async addTask(taskText, priority = 'normal') {
    const operationStart = performance.now();

    try {
        // バリデーション
        if (!taskText || taskText.trim() === '') {
            throw new Error('タスクテキストが必要です');
        }

        // 新しいタスクの作成
        const newTask = {
            id: this.generateOptimizedId(),
            text: taskText.trim(),
            priority: priority,
```

```
            completed: false,
            createdAt: new Date().toISOString(),
            updatedAt: new Date().toISOString()
        };

        // メモリキャッシュの更新
        const currentTasks = this.memoryManager.get('allTasks') || [];
        currentTasks.push(newTask);
        this.memoryManager.set('allTasks', currentTasks);

        // 遅延保存
        this.debouncedManager.debouncedSave('tasks', currentTasks);

        // UI の即座更新
        this.addTaskToUIOptimized(newTask);

        const operationTime = performance.now() - operationStart;
        console.log(`タスク追加: ${operationTime.toFixed(2)}ms`);

        return newTask;

    } catch (error) {
        console.error('タスク追加エラー:', error);
        throw error;
    }
}

// 高性能なタスク更新
async updateTask(taskId, updates) {
    const operationStart = performance.now();

    try {
        const tasks = this.memoryManager.get('allTasks') || [];
        const taskIndex = tasks.findIndex(task => task.id === taskId);

        if (taskIndex === -1) {
            throw new Error('タスクが見つかりません');
        }

        // タスクの更新
        tasks[taskIndex] = {
            ...tasks[taskIndex],
            ...updates,
            updatedAt: new Date().toISOString()
        };

        // キャッシュの更新
        this.memoryManager.set('allTasks', tasks);

        // バッチ処理による遅延保存
        this.debouncedManager.addToBatch({
            type: 'save',
            key: 'tasks',
            data: tasks
        });
```

```
        // UI の効率的な部分更新
        this.updateTaskElementOptimized(taskId, tasks[taskIndex]);

        const operationTime = performance.now() - operationStart;
        console.log(`タスク更新: ${operationTime.toFixed(2)}ms`);

        return tasks[taskIndex];

    } catch (error) {
        console.error('タスク更新エラー:', error);
        throw error;
    }
}

// 高性能なタスク削除
async deleteTask(taskId) {
    const operationStart = performance.now();

    try {
        const tasks = this.memoryManager.get('allTasks') || [];
        const taskIndex = tasks.findIndex(task => task.id === taskId);

        if (taskIndex === -1) {
            throw new Error('タスクが見つかりません');
        }

        const deletedTask = tasks[taskIndex];

        // 論理削除（パフォーマンス向上）
        tasks[taskIndex].deleted = true;
        tasks[taskIndex].deletedAt = new Date().toISOString();

        // キャッシュの更新
        this.memoryManager.set('allTasks', tasks);

        // 遅延保存
        this.debouncedManager.debouncedSave('tasks', tasks);

        // UI からの即座削除
        this.removeTaskFromUIOptimized(taskId);

        const operationTime = performance.now() - operationStart;
        console.log(`タスク削除: ${operationTime.toFixed(2)}ms`);

        return deletedTask;

    } catch (error) {
        console.error('タスク削除エラー:', error);
        throw error;
    }
}

// 最適化された検索機能
searchTasks(query, options = {}) {
```

```
        const searchStart = performance.now();

        const tasks = this.memoryManager.get('allTasks') || [];
        const {
            caseSensitive = false,
            maxResults = 100,
            includeCompleted = true
        } = options;

        const searchTerm = caseSensitive ? query : query.toLowerCase();

        const results = tasks.filter(task => {
            if (task.deleted) return false;
            if (!includeCompleted && task.completed) return false;

            const taskText = caseSensitive ? task.text : task.text.toLowerCase();
            return taskText.includes(searchTerm);
        }).slice(0, maxResults);

        const searchTime = performance.now() - searchStart;
        console.log(`検索完了: ${results.length}件 (${searchTime.toFixed(2)}ms)`);

        return results;
    }

    // UI最適化メソッド
    addTaskToUIOptimized(task) {
        const container = document.getElementById('taskList');
        const taskElement = this.createTaskElementOptimized(task);

        // DocumentFragment を使用して効率的な DOM 操作
        const fragment = document.createDocumentFragment();
        fragment.appendChild(taskElement);
        container.appendChild(fragment);
    }

    createTaskElementOptimized(task) {
        // テンプレートクローンによる高速な要素作成
        const template = document.getElementById('taskTemplate');
        const clone = template.content.cloneNode(true);

        // 必要な部分のみを更新
        const taskElement = clone.querySelector('.task-item');
        taskElement.dataset.id = task.id;

        const textElement = clone.querySelector('.task-text');
        textElement.textContent = task.text;

        const checkbox = clone.querySelector('.task-checkbox');
        checkbox.checked = task.completed;

        return clone;
    }

    updateTaskElementOptimized(taskId, task) {
```

```
        const element = document.querySelector(`[data-id="${taskId}"]`);
        if (!element) return;

        // 必要な部分のみを更新（全体の再描画を避ける）
        const textElement = element.querySelector('.task-text');
        const checkbox = element.querySelector('.task-checkbox');

        if (textElement.textContent !== task.text) {
            textElement.textContent = task.text;
        }

        if (checkbox.checked !== task.completed) {
            checkbox.checked = task.completed;
        }

        // クラスの効率的な更新
        element.classList.toggle('completed', task.completed);
    }

    removeTaskFromUIOptimized(taskId) {
        const element = document.querySelector(`[data-id="${taskId}"]`);
        if (element) {
            // フェードアウトアニメーション
            element.style.opacity = '0';
            element.style.transition = 'opacity 0.3s ease';

            setTimeout(() => {
                element.remove();
            }, 300);
        }
    }

    // パフォーマンス最適化の実行
    optimizePerformance() {
        console.log('パフォーマンス最適化を実行中...');

        // メモリクリーンアップ
        this.memoryManager.cleanupMemory();

        // 古いキャッシュの削除
        this.cachedStorage.cleanupOldData();

        // 遅延保存の強制実行
        this.debouncedManager.flushAll();

        // ガベージコレクションのヒント
        if (window.gc) {
            window.gc();
        }

        console.log('パフォーマンス最適化完了');
    }

    // 最適化されたID生成
    generateOptimizedId() {
```

```
        // より効率的なID生成（crypto API利用）
        if (crypto && crypto.randomUUID) {
            return crypto.randomUUID();
        }

        // フォールバック：高速な疑似ランダムID
        return `task_${Date.now()}_${Math.random().toString(36).substr(2, 6)}`;
    }

    // 自動保存の設定
    setupAutoSave() {
        setInterval(() => {
            this.debouncedManager.flushAll();
        }, this.config.autoSaveInterval);

        // ページ離脱時の保存
        window.addEventListener('beforeunload', () => {
            this.debouncedManager.flushAll();
        });
    }

    // パフォーマンス統計の取得
    getPerformanceStats() {
        return {
            monitor: this.performanceMonitor.generateReport(),
            memory: this.memoryManager.getMemoryStats(),
            cache: this.cachedStorage.getCacheStats ? this.cachedStorage.getCacheStats() : null
        };
    }
}

// アプリケーションの初期化
document.addEventListener('DOMContentLoaded', () => {
    // パフォーマンス測定開始
    const initStart = performance.now();

    try {
        const app = new HighPerformanceTodoApp();

        const initTime = performance.now() - initStart;
        console.log(`アプリ初期化時間: ${initTime.toFixed(2)}ms`);

        // 開発時のパフォーマンス監視
        if (process.env.NODE_ENV === 'development') {
            setInterval(() => {
                const stats = app.getPerformanceStats();
                console.table(stats.monitor);
            }, 10000);
        }

    } catch (error) {
        console.error('アプリケーション初期化失敗:', error);
    }
});
```

このlocalStorageパフォーマンス最適化の実装により、**大量データを扱う場合でも高速で応答性の良い**TodoListアプリケーションを構築できます。

> 参考：さらなる詳細は フロントエンド状態管理、DOM イベント処理 をご覧ください。