データ整合性管理

> 関連資料: CRUD パターン詳解 | localStorage API詳解 | try-catch詳解

1. 基本（1分で理解）

データ整合性とは、アプリケーション内のデータが**常に正しく一貫した状態**を保つことです。不整合データが存在すると、アプリケーションが予期しない動作をしたり、ユーザーに混乱を与えたりします。

整合性が重要な理由

- **ユーザー体験の向上**：データの矛盾によるエラーを防ぐ
- **アプリケーションの安定性**：予期しないクラッシュを回避
- **データの信頼性**：正確な情報を提供

```
// 不整合の例：存在しないタスクを更新しようとする
const tasks = [
    { id: 1, text: "買い物", completed: false }
];

// 危険：存在しないIDを更新
updateTask(999, { completed: true }); // エラーが発生する可能性

// 安全：事前に存在確認
if (taskExists(999)) {
    updateTask(999, { completed: true });
} else {
    console.error("タスクが見つかりません");
}
```

2. 詳細（3分で習得）

2.1 データバリデーション

入力データの検証により、不正なデータの保存を防ぎます。

```
class TaskValidator {
    static validateTask(task) {
        const errors = [];

        // 必須フィールドの検証
        if (!task.text || task.text.trim() === '') {
            errors.push('タスクテキストは必須です');
        }

        // データ型の検証
        if (typeof task.completed !== 'boolean') {
            errors.push('完了状態は真偽値である必要があります');
        }

        // 値の範囲検証
        if (task.text && task.text.length > 500) {
            errors.push('タスクテキストは500文字以内である必要があります');
        }

        // 優先度の検証
        const validPriorities = ['low', 'normal', 'high'];
        if (task.priority && !validPriorities.includes(task.priority)) {
            errors.push('無効な優先度です');
        }

        // 日付の検証
        if (task.dueDate && isNaN(new Date(task.dueDate).getTime())) {
            errors.push('無効な期限日です');
        }

        return {
            isValid: errors.length === 0,
            errors: errors
        };
    }

    static validateTaskList(tasks) {
        // 配列の検証
        if (!Array.isArray(tasks)) {
            return { isValid: false, errors: ['タスクリストは配列である必要があります'] };
        }

        const errors = [];
        const ids = new Set();

        tasks.forEach((task, index) => {
            // 各タスクの個別検証
            const taskValidation = this.validateTask(task);
            if (!taskValidation.isValid) {
```

```
                errors.push(`タスク${index + 1}: ${taskValidation.errors.join(', ')}`);
            }

            // ID重複の検証
            if (task.id) {
                if (ids.has(task.id)) {
                    errors.push(`重複するID: ${task.id}`);
                }
                ids.add(task.id);
            }
        });

        return {
            isValid: errors.length === 0,
            errors: errors
        };
    }
}
```

2.2 状態遷移管理

不正な状態変更を防ぐための制御機構です。

```
class TaskStateManager {
    constructor() {
        // 有効な状態遷移の定義
        this.allowedTransitions = {
            'draft': ['active', 'cancelled'],
            'active': ['completed', 'cancelled', 'paused'],
            'paused': ['active', 'cancelled'],
            'completed': ['active'],  // 完了を取り消すことは可能
            'cancelled': ['draft']    // 取り消しから下書きに戻すことは可能
        };
    }

    canTransition(fromState, toState) {
        const allowed = this.allowedTransitions[fromState] || [];
        return allowed.includes(toState);
    }

    validateStateTransition(task, newState) {
        const currentState = task.status || 'draft';

        if (!this.canTransition(currentState, newState)) {
            throw new Error(
                `無効な状態遷移: ${currentState} から ${newState} への変更はできません`
            );
        }

        // 特定の状態変更に対する追加検証
        if (newState === 'completed' && !task.text.trim()) {
            throw new Error('空のタスクは完了できません');
```

```
        }

        if (newState === 'active' && task.dueDate && new Date(task.dueDate) < new Date()) {
            console.warn('期限切れのタスクをアクティブにしています');
        }

        return true;
    }
}
```

2.3 データ同期管理

複数のデータソース間での整合性を保ちます。

```
class DataSyncManager {
    constructor(storageKey) {
        this.storageKey = storageKey;
        this.memoryCache = null;
        this.lastSyncTime = null;
        this.version = 0;
    }

    // メモリとlocalStorageの同期
    syncWithStorage() {
        try {
            const stored = localStorage.getItem(this.storageKey);
            const storedData = stored ? JSON.parse(stored) : [];

            // バージョン管理による競合検出
            const storedVersion = localStorage.getItem(`${this.storageKey}_version`) || 0;

            if (this.memoryCache && this.version !== parseInt(storedVersion)) {
                console.warn('データの競合が検出されました。最新データで更新します。');
                this.handleConflict(this.memoryCache, storedData);
            }

            this.memoryCache = storedData;
            this.version = parseInt(storedVersion);
            this.lastSyncTime = new Date();

            return storedData;
        } catch (error) {
            console.error('データ同期エラー:', error);
            // エラー時はメモリキャッシュを優先
            return this.memoryCache || [];
        }
    }

    // データの保存とバージョン更新
    saveToStorage(data) {
        try {
            // 保存前の整合性チェック
```

```
            const validation = TaskValidator.validateTaskList(data);
            if (!validation.isValid) {
                throw new Error(`データ整合性エラー: ${validation.errors.join(', ')}`);
            }

            // アトミックな保存（バージョンとデータを同時更新）
            this.version++;
            localStorage.setItem(this.storageKey, JSON.stringify(data));
            localStorage.setItem(`${this.storageKey}_version`, this.version.toString());

            this.memoryCache = data;
            this.lastSyncTime = new Date();

            // 他のタブに変更を通知
            this.notifyOtherTabs(data);

        } catch (error) {
            console.error('データ保存エラー:', error);
            throw error;
        }
    }

    // 競合解決
    handleConflict(localData, storageData) {
        // 簡単な競合解決：最新のタイムスタンプを優先
        const mergedData = this.mergeDataByTimestamp(localData, storageData);
        this.memoryCache = mergedData;
        return mergedData;
    }

    mergeDataByTimestamp(data1, data2) {
        const merged = new Map();

        // data1のタスクを追加
        data1.forEach(task => {
            merged.set(task.id, task);
        });

        // data2のタスクで新しいものがあれば更新
        data2.forEach(task => {
            const existing = merged.get(task.id);
            if (!existing || new Date(task.updatedAt) > new Date(existing.updatedAt)) {
                merged.set(task.id, task);
            }
        });

        return Array.from(merged.values());
    }

    // 他のタブへの通知
    notifyOtherTabs(data) {
        // Custom event for same-origin tabs
        const event = new CustomEvent('dataSync', {
            detail: { data, version: this.version, timestamp: new Date().toISOString() }
        });
```

```
        window.dispatchEvent(event);

        // BroadcastChannel API for modern browsers
        if (typeof BroadcastChannel !== 'undefined') {
            const channel = new BroadcastChannel(`${this.storageKey}_sync`);
            channel.postMessage({
                type: 'dataUpdate',
                data: data,
                version: this.version,
                timestamp: new Date().toISOString()
            });
        }
    }
}
```

2.4 トランザクション管理

複数の操作を原子的に実行し、失敗時には全体をロールバックします。

```
class TaskTransaction {
    constructor(dataManager) {
        this.dataManager = dataManager;
        this.operations = [];
        this.backup = null;
    }

    begin() {
        // 現在の状態をバックアップ
        this.backup = JSON.parse(JSON.stringify(this.dataManager.getTasks()));
        this.operations = [];
        return this;
    }

    addOperation(operation) {
        this.operations.push(operation);
        return this;
    }

    async execute() {
        try {
            // すべての操作を順次実行
            for (const operation of this.operations) {
                await operation();
            }

            // 成功時：変更をコミット
            this.commit();
            return { success: true };

        } catch (error) {
            // 失敗時：ロールバック
            this.rollback();
```

```
            throw new Error(`トランザクション失敗: ${error.message}`);
        }
    }

    commit() {
        // バックアップを破棄
        this.backup = null;
        this.operations = [];
    }

    rollback() {
        if (this.backup) {
            // データを元の状態に復元
            this.dataManager.setTasks(this.backup);
            this.backup = null;
        }
        this.operations = [];
    }
}

// 使用例：複数タスクの一括更新
async function bulkUpdateTasks(taskUpdates) {
    const transaction = new TaskTransaction(taskManager);

    try {
        transaction.begin();

        // 各更新操作をトランザクションに追加
        taskUpdates.forEach(update => {
            transaction.addOperation(async () => {
                await taskManager.updateTask(update.id, update.changes);
            });
        });

        await transaction.execute();
        console.log('一括更新が完了しました');

    } catch (error) {
        console.error('一括更新に失敗しました:', error.message);
    }
}
```

3. 深掘りコラム

3.1 ACID原則とフロントエンド

データベースのACID原則をフロントエンドに適用する方法：

```
class ACIDCompliantTaskManager {
    constructor() {
        this.dataManager = new DataSyncManager('tasks');
        this.lockManager = new LockManager();
    }

    // Atomicity（原子性）：全部成功か全部失敗
    async atomicOperation(operations) {
        const transaction = new TaskTransaction(this.dataManager);
        return transaction.begin().addOperations(operations).execute();
    }

    // Consistency（一貫性）：データの整合性を保つ
    async saveWithConsistencyCheck(tasks) {
        const validation = TaskValidator.validateTaskList(tasks);
        if (!validation.isValid) {
            throw new Error(`一貫性エラー: ${validation.errors.join(', ')}`);
        }

        // 参照整合性のチェック（例：親子関係のあるタスク）
        this.validateReferentialIntegrity(tasks);

        return this.dataManager.saveToStorage(tasks);
    }

    // Isolation（分離性）：操作の分離
    async isolatedUpdate(taskId, updateFn) {
        const lock = await this.lockManager.acquireLock(taskId);
        try {
            const task = this.dataManager.getTaskById(taskId);
            const updatedTask = updateFn(task);
            await this.saveWithConsistencyCheck([updatedTask]);
            return updatedTask;
        } finally {
            this.lockManager.releaseLock(taskId, lock);
        }
    }

    // Durability（永続性）：データの永続化
    async durableSave(tasks) {
        // 複数の保存先に書き込み
        const promises = [
            this.dataManager.saveToStorage(tasks),
            this.saveToIndexedDB(tasks),
            this.syncToServer(tasks)
        ];

        await Promise.all(promises);
    }
}
```

3.2 イベントソーシング

すべての変更を履歴として記録し、データの完全な追跡を可能にします：

```
class TaskEventStore {
    constructor() {
        this.events = this.loadEvents();
        this.snapshots = new Map();
    }

    // イベントの記録
    recordEvent(event) {
        const eventWithTimestamp = {
            ...event,
            id: this.generateEventId(),
            timestamp: new Date().toISOString(),
            version: this.events.length + 1
        };

        this.events.push(eventWithTimestamp);
        this.saveEvents();

        return eventWithTimestamp;
    }

    // イベントからタスクリストを再構築
    reconstructTaskList(upToVersion = null) {
        const targetEvents = upToVersion
            ? this.events.slice(0, upToVersion)
            : this.events;

        const tasks = new Map();

        targetEvents.forEach(event => {
            switch (event.type) {
                case 'TASK_CREATED':
                    tasks.set(event.taskId, {
                        id: event.taskId,
                        text: event.text,
                        completed: false,
                        createdAt: event.timestamp,
                        updatedAt: event.timestamp
                    });
                    break;

                case 'TASK_UPDATED':
                    const existingTask = tasks.get(event.taskId);
                    if (existingTask) {
                        tasks.set(event.taskId, {
                            ...existingTask,
                            ...event.changes,
                            updatedAt: event.timestamp
                        });
                    }
                    break;
```

```
                case 'TASK_DELETED':
                    tasks.delete(event.taskId);
                    break;
            }
        });

        return Array.from(tasks.values());
    }

    // スナップショットの作成（パフォーマンス向上）
    createSnapshot(version) {
        const taskList = this.reconstructTaskList(version);
        this.snapshots.set(version, {
            tasks: taskList,
            timestamp: new Date().toISOString()
        });

        // 古いスナップショットの削除
        this.cleanupOldSnapshots();
    }

    // 時点復元
    restoreToVersion(version) {
        const tasks = this.reconstructTaskList(version);
        return tasks;
    }
}
```

3.3 リアルタイム同期

WebSocketやSSEを使用したリアルタイムデータ同期：

```
class RealtimeTaskSync {
    constructor(taskManager) {
        this.taskManager = taskManager;
        this.websocket = null;
        this.reconnectAttempts = 0;
        this.maxReconnectAttempts = 5;
    }

    connect() {
        this.websocket = new WebSocket('ws://localhost:8080/tasks');

        this.websocket.onopen = () => {
            console.log('リアルタイム同期が開始されました');
            this.reconnectAttempts = 0;
        };

        this.websocket.onmessage = (event) => {
            const data = JSON.parse(event.data);
            this.handleRemoteUpdate(data);
```

```
        };

        this.websocket.onclose = () => {
            console.log('接続が閉じられました');
            this.attemptReconnect();
        };

        this.websocket.onerror = (error) => {
            console.error('WebSocket エラー:', error);
        };
    }

    handleRemoteUpdate(data) {
        switch (data.type) {
            case 'TASK_UPDATED':
                // 競合解決
                const localTask = this.taskManager.getTaskById(data.task.id);
                const resolvedTask = this.resolveConflict(localTask, data.task);
                this.taskManager.updateTaskSilently(data.task.id, resolvedTask);
                break;

            case 'TASK_CREATED':
                this.taskManager.addTaskSilently(data.task);
                break;

            case 'TASK_DELETED':
                this.taskManager.deleteTaskSilently(data.task.id);
                break;
        }

        // UI の更新通知
        this.notifyUIUpdate(data);
    }

    resolveConflict(localTask, remoteTask) {
        // Last-Write-Wins 戦略
        const localTime = new Date(localTask.updatedAt);
        const remoteTime = new Date(remoteTask.updatedAt);

        return remoteTime > localTime ? remoteTask : localTask;
    }
}
```

4. 実践応用

4.1 完全なデータ整合性を持つTodoアプリ

```
class ConsistentTodoApp {
    constructor() {
        this.validator = new TaskValidator();
        this.stateManager = new TaskStateManager();
        this.dataSync = new DataSyncManager('todoTasks');
        this.eventStore = new TaskEventStore();

        this.setupEventListeners();
        this.initializeApp();
    }

    async initializeApp() {
        try {
            // データの整合性チェックと復旧
            await this.validateAndRepairData();

            // UI の初期化
            await this.loadAndDisplayTasks();

            // マルチタブ同期の設定
            this.setupMultiTabSync();

        } catch (error) {
            this.handleInitializationError(error);
        }
    }

    async validateAndRepairData() {
        try {
            const tasks = this.dataSync.syncWithStorage();
            const validation = this.validator.validateTaskList(tasks);

            if (!validation.isValid) {
                console.warn('データ整合性エラーが検出されました:', validation.errors);

                // 自動修復を試行
                const repairedTasks = await this.repairData(tasks, validation.errors);
                this.dataSync.saveToStorage(repairedTasks);

                this.showNotification('データの不整合を自動修復しました', 'warning');
            }
        } catch (error) {
            console.error('データ検証エラー:', error);
            // フォールバック：空のタスクリストで開始
            this.dataSync.saveToStorage([]);
        }
    }

    async repairData(tasks, errors) {
        const repairedTasks = [];
        const usedIds = new Set();

        tasks.forEach((task, index) => {
            try {
```

```
            // 必須フィールドの修復
            const repairedTask = {
                id: task.id || `repaired_${Date.now()}_${index}`,
                text: task.text || '（修復されたタスク）',
                completed: Boolean(task.completed),
                priority: ['low', 'normal', 'high'].includes(task.priority) ? task.priority : 'normal',
                createdAt: task.createdAt || new Date().toISOString(),
                updatedAt: task.updatedAt || new Date().toISOString()
            };

            // ID重複の修復
            if (usedIds.has(repairedTask.id)) {
                repairedTask.id = `${repairedTask.id}_dedup_${Date.now()}`;
            }
            usedIds.add(repairedTask.id);

            // 個別検証
            const taskValidation = this.validator.validateTask(repairedTask);
            if (taskValidation.isValid) {
                repairedTasks.push(repairedTask);
            }

        } catch (error) {
            console.warn(`タスク ${index} の修復に失敗:`, error);
        }
    });

    return repairedTasks;
}

async createTask(taskText, priority = 'normal') {
    try {
        // 事前検証
        const newTask = {
            id: this.generateId(),
            text: taskText.trim(),
            priority: priority,
            completed: false,
            createdAt: new Date().toISOString(),
            updatedAt: new Date().toISOString()
        };

        const validation = this.validator.validateTask(newTask);
        if (!validation.isValid) {
            throw new Error(validation.errors.join(', '));
        }

        // トランザクション開始
        const transaction = new TaskTransaction(this);
        transaction.begin();

        try {
            // データ追加
            const currentTasks = this.dataSync.syncWithStorage();
            const updatedTasks = [...currentTasks, newTask];
```

```
            // 整合性チェック
            const listValidation = this.validator.validateTaskList(updatedTasks);
            if (!listValidation.isValid) {
                throw new Error(`リスト整合性エラー: ${listValidation.errors.join(', ')}`);
            }

            // 保存
            this.dataSync.saveToStorage(updatedTasks);

            // イベント記録
            this.eventStore.recordEvent({
                type: 'TASK_CREATED',
                taskId: newTask.id,
                text: newTask.text,
                priority: newTask.priority
            });

            transaction.commit();

            // UI更新
            this.addTaskToUI(newTask);
            this.showNotification('タスクが追加されました', 'success');

            return newTask;

        } catch (error) {
            transaction.rollback();
            throw error;
        }

    } catch (error) {
        console.error('タスク作成エラー:', error);
        this.showNotification(error.message, 'error');
        throw error;
    }
}

async updateTask(taskId, changes) {
    try {
        const currentTasks = this.dataSync.syncWithStorage();
        const taskIndex = currentTasks.findIndex(task => task.id === taskId);

        if (taskIndex === -1) {
            throw new Error(`タスク ID: ${taskId} が見つかりません`);
        }

        const currentTask = currentTasks[taskIndex];

        // 状態遷移の検証
        if (changes.status) {
            this.stateManager.validateStateTransition(currentTask, changes.status);
        }

        // 更新データの作成
```

```
        const updatedTask = {
            ...currentTask,
            ...changes,
            updatedAt: new Date().toISOString()
        };

        // 検証
        const validation = this.validator.validateTask(updatedTask);
        if (!validation.isValid) {
            throw new Error(validation.errors.join(', '));
        }

        // 更新の実行
        const newTaskList = [...currentTasks];
        newTaskList[taskIndex] = updatedTask;

        this.dataSync.saveToStorage(newTaskList);

        // イベント記録
        this.eventStore.recordEvent({
            type: 'TASK_UPDATED',
            taskId: taskId,
            changes: changes,
            previousState: currentTask
        });

        // UI更新
        this.updateTaskInUI(taskId, updatedTask);
        this.showNotification('タスクが更新されました', 'success');

        return updatedTask;

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

async deleteTask(taskId) {
    try {
        const currentTasks = this.dataSync.syncWithStorage();
        const taskIndex = currentTasks.findIndex(task => task.id === taskId);

        if (taskIndex === -1) {
            throw new Error(`タスク ID: ${taskId} が見つかりません`);
        }

        const taskToDelete = currentTasks[taskIndex];
        const updatedTasks = currentTasks.filter(task => task.id !== taskId);

        this.dataSync.saveToStorage(updatedTasks);

        // イベント記録
        this.eventStore.recordEvent({
```

```
            type: 'TASK_DELETED',
            taskId: taskId,
            deletedTask: taskToDelete
        });

        // UI更新
        this.removeTaskFromUI(taskId);
        this.showNotification('タスクが削除されました', 'success');

        return taskToDelete;

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

setupMultiTabSync() {
    // Storage event for cross-tab synchronization
    window.addEventListener('storage', (e) => {
        if (e.key === 'todoTasks') {
            this.handleExternalDataChange(e.newValue);
        }
    });

    // Custom event for same-tab updates
    window.addEventListener('dataSync', (e) => {
        this.handleDataSyncEvent(e.detail);
    });
}

handleExternalDataChange(newData) {
    try {
        const tasks = newData ? JSON.parse(newData) : [];
        const validation = this.validator.validateTaskList(tasks);

        if (validation.isValid) {
            this.renderTaskList(tasks);
            this.showNotification('他のタブでの変更を同期しました', 'info');
        } else {
            console.warn('他のタブからの不正なデータを検出:', validation.errors);
        }
    } catch (error) {
        console.error('外部データ変更の処理エラー:', error);
    }
}

generateId() {
    return `task_${Date.now()}_${Math.random().toString(36).substr(2, 9)}`;
}

showNotification(message, type) {
    // 通知システムの実装
    const notification = document.createElement('div');
```

```
        notification.className = `notification ${type}`;
        notification.textContent = message;
        document.body.appendChild(notification);

        setTimeout(() => notification.remove(), 3000);
    }
}

// アプリケーションの初期化
document.addEventListener('DOMContentLoaded', () => {
    try {
        const app = new ConsistentTodoApp();

        // グローバルエラーハンドリング
        window.addEventListener('error', (e) => {
            console.error('アプリケーションエラー:', e.error);
            app.showNotification('予期しないエラーが発生しました', 'error');
        });

        window.addEventListener('unhandledrejection', (e) => {
            console.error('未処理のPromise拒否:', e.reason);
            app.showNotification('非同期処理でエラーが発生しました', 'error');
        });

    } catch (error) {
        console.error('アプリケーション初期化エラー:', error);
        document.body.innerHTML = '<p>アプリケーションの初期化に失敗しました。ページを再読み込みしてください。</p>';
    }
});
```

4.2 デバッグ用整合性チェッカー

```
class DataIntegrityChecker {
    constructor(taskManager) {
        this.taskManager = taskManager;
    }

    performFullCheck() {
        const results = {
            passed: 0,
            failed: 0,
            warnings: 0,
            details: []
        };

        try {
            // 基本整合性チェック
            this.checkBasicIntegrity(results);

            // 参照整合性チェック
            this.checkReferentialIntegrity(results);
```

```
        // ビジネスルールチェック
        this.checkBusinessRules(results);

        // パフォーマンスチェック
        this.checkPerformance(results);

    } catch (error) {
        results.details.push({
            type: 'error',
            message: `整合性チェック中にエラーが発生: ${error.message}`
        });
        results.failed++;
    }

    return results;
}

checkBasicIntegrity(results) {
    const tasks = this.taskManager.getTasks();

    // ID重複チェック
    const ids = tasks.map(task => task.id);
    const uniqueIds = new Set(ids);
    if (ids.length !== uniqueIds.size) {
        results.details.push({
            type: 'error',
            message: 'ID重複が検出されました'
        });
        results.failed++;
    } else {
        results.passed++;
    }

    // 必須フィールドチェック
    tasks.forEach((task, index) => {
        if (!task.id || !task.text || task.completed === undefined) {
            results.details.push({
                type: 'error',
                message: `タスク${index + 1}に必須フィールドが不足しています`
            });
            results.failed++;
        } else {
            results.passed++;
        }
    });
}

generateReport(results) {
    const report = {
        summary: {
            total: results.passed + results.failed + results.warnings,
            passed: results.passed,
            failed: results.failed,
            warnings: results.warnings,
            success: results.failed === 0
```

```
            },
            details: results.details,
            timestamp: new Date().toISOString()
        };

        console.log('データ整合性レポート:', report);
        return report;
    }
}
```

このデータ整合性管理の実装により、**堅牢で信頼性の高い**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 localStorage パフォーマンス、フロントエンド状態管理をご覧ください。