CRUDパターン詳解

> 関連資料: データ構造設計 | localStorage API詳解 | データ整合性管理

1. 基本（1分で理解）

CRUDパターンとは、データ操作の基本的な4つの操作を体系化したパターンです：

- **C**reate（作成）：新しいデータを追加
- **R**ead（読み取り）：既存データを取得・表示
- **U**pdate（更新）：既存データを変更
- **D**elete（削除）：データを削除

これらの操作を分離して設計することで、**保守性が高く**、**バグが少ない**アプリケーションを構築できます。

```
// 基本的なCRUD操作の例
const todoManager = {
    create: (task) => { /* タスクを追加 */ },
    read: () => { /* タスク一覧を取得 */ },
    update: (id, changes) => { /* タスクを更新 */ },
    delete: (id) => { /* タスクを削除 */ }
};
```

2. 詳細（3分で習得）

2.1 Create（作成）操作

新しいデータを生成し、ストレージに保存する操作です。

```
class TodoCRUD {
    constructor() {
        this.storageKey = 'todoList';
    }

    // Create: 新しいタスクを作成
    createTask(taskText, priority = 'normal') {
        try {
            // 入力バリデーション
            if (!taskText || taskText.trim() === '') {
                throw new Error('タスクテキストが空です');
            }

            // 既存データの取得
            const todos = this.readTasks();

            // 新しいタスクオブジェクトの作成
            const newTask = {
                id: this.generateId(),
                text: taskText.trim(),
                priority: priority,
                completed: false,
                createdAt: new Date().toISOString(),
                updatedAt: new Date().toISOString()
            };

            // データの追加と保存
            todos.push(newTask);
            this.saveTasks(todos);

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

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

2.2 Read（読み取り）操作

ストレージからデータを取得し、必要に応じてフィルタリングや並び替えを行います。

```
// Read: タスクの読み取り操作
readTasks(filter = null) {
    try {
        const stored = localStorage.getItem(this.storageKey);
```

```
        let todos = stored ? JSON.parse(stored) : [];

        // フィルタリング機能
        if (filter) {
            switch (filter.type) {
                case 'completed':
                    todos = todos.filter(task => task.completed === filter.value);
                    break;
                case 'priority':
                    todos = todos.filter(task => task.priority === filter.value);
                    break;
                case 'search':
                    todos = todos.filter(task =>
                        task.text.toLowerCase().includes(filter.value.toLowerCase())
                    );
                    break;
            }
        }

        // デフォルトのソート（作成日時の降順）
        todos.sort((a, b) => new Date(b.createdAt) - new Date(a.createdAt));

        return todos;
    } catch (error) {
        console.error('タスク読み取りエラー:', error);
        return [];
    }
}

// 特定のタスクを取得
readTaskById(id) {
    const todos = this.readTasks();
    return todos.find(task => task.id === id) || null;
}
```

2.3 Update（更新）操作

既存データの特定フィールドを変更し、変更履歴を記録します。

```
// Update: タスクの更新操作
updateTask(id, updates) {
    try {
        const todos = this.readTasks();
        const taskIndex = todos.findIndex(task => task.id === id);

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

        // 更新前の状態を保存（Undo機能用）
        const originalTask = { ...todos[taskIndex] };
```

```
        // 更新可能なフィールドのみを変更
        const allowedFields = ['text', 'priority', 'completed'];
        const validUpdates = {};

        for (const [key, value] of Object.entries(updates)) {
            if (allowedFields.includes(key)) {
                validUpdates[key] = value;
            }
        }

        // バリデーション
        if (validUpdates.text !== undefined && !validUpdates.text.trim()) {
            throw new Error('タスクテキストは空にできません');
        }

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

        this.saveTasks(todos);

        // 変更履歴の記録
        this.recordChange('update', { original: originalTask, updated: todos[taskIndex] });

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

// 完了状態の切り替え（よく使用される更新操作）
toggleTaskCompletion(id) {
    const task = this.readTaskById(id);
    if (!task) {
        throw new Error(`ID: ${id} のタスクが見つかりません`);
    }

    return this.updateTask(id, { completed: !task.completed });
}
```

2.4 Delete（削除）操作

データを安全に削除し、必要に応じて論理削除も実装します。

```
// Delete: タスクの削除操作
deleteTask(id, permanent = false) {
    try {
        const todos = this.readTasks();
```

```
        const taskIndex = todos.findIndex(task => task.id === id);

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

        const deletedTask = todos[taskIndex];

        if (permanent) {
            // 物理削除：完全にデータを削除
            todos.splice(taskIndex, 1);
        } else {
            // 論理削除：削除フラグを設定
            todos[taskIndex] = {
                ...todos[taskIndex],
                deleted: true,
                deletedAt: new Date().toISOString()
            };
        }

        this.saveTasks(todos);

        // 削除履歴の記録
        this.recordChange('delete', { deleted: deletedTask, permanent });

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

// ゴミ箱機能：論理削除されたタスクを取得
getDeletedTasks() {
    const todos = this.readTasks();
    return todos.filter(task => task.deleted === true);
}

// 完全削除の実行
permanentDelete(id) {
    return this.deleteTask(id, true);
}
```

3. 深掘りコラム

3.1 CRUDパターンの設計原則

Single Responsibility Principle（単一責任原則）に基づき、各操作は一つの責任のみを持つべきです：

```
class AdvancedTodoCRUD extends TodoCRUD {
    constructor() {
        super();
        this.validator = new TaskValidator();
        this.storage = new StorageManager();
        this.history = new ChangeHistory();
    }

    // バリデーションの分離
    validateTask(task) {
        return this.validator.validate(task);
    }

    // ストレージ操作の分離
    saveTasks(tasks) {
        return this.storage.save(this.storageKey, tasks);
    }

    // 変更履歴の分離
    recordChange(operation, data) {
        return this.history.record(operation, data);
    }
}
```

3.2 楽観的ロック vs 悲観的ロック

同時編集問題を解決するためのアプローチ：

```
// 楽観的ロック：バージョン管理による競合検出
class OptimisticLockingCRUD extends TodoCRUD {
    updateTask(id, updates) {
        const currentTask = this.readTaskById(id);

        // バージョンチェック
        if (updates.version && updates.version !== currentTask.version) {
            throw new Error('他のユーザーによって更新されています。最新データを再読み込みしてください。');
        }

        // バージョンを更新
        updates.version = (currentTask.version || 0) + 1;

        return super.updateTask(id, updates);
    }
}

// マルチタブ環境での同期
```

```
class MultiTabCRUD extends TodoCRUD {
    constructor() {
        super();
        // Storage Event でタブ間同期
        window.addEventListener('storage', (e) => {
            if (e.key === this.storageKey) {
                this.onDataChanged(e.newValue);
            }
        });
    }

    onDataChanged(newData) {
        // UI の自動更新
        const event = new CustomEvent('tasksUpdated', {
            detail: { tasks: JSON.parse(newData || '[]') }
        });
        window.dispatchEvent(event);
    }
}
```

3.3 Batch Operations（一括操作）

パフォーマンス向上のための一括処理：

```
class BatchCRUD extends TodoCRUD {
    // 一括作成
    createMultipleTasks(taskTexts) {
        const todos = this.readTasks();
        const newTasks = taskTexts.map(text => ({
            id: this.generateId(),
            text: text.trim(),
            priority: 'normal',
            completed: false,
            createdAt: new Date().toISOString(),
            updatedAt: new Date().toISOString()
        }));

        todos.push(...newTasks);
        this.saveTasks(todos);

        return newTasks;
    }

    // 一括更新
    updateMultipleTasks(updates) {
        const todos = this.readTasks();
        let modifiedCount = 0;

        for (const { id, changes } of updates) {
            const taskIndex = todos.findIndex(task => task.id === id);
            if (taskIndex !== -1) {
                todos[taskIndex] = {
```

```
                    ...todos[taskIndex],
                    ...changes,
                    updatedAt: new Date().toISOString()
                };
                modifiedCount++;
            }
        }

        this.saveTasks(todos);
        return { modifiedCount };
    }

    // 条件に基づく一括削除
    deleteTasksByCondition(condition) {
        const todos = this.readTasks();
        const toDelete = todos.filter(condition);
        const remaining = todos.filter(task => !condition(task));

        this.saveTasks(remaining);
        return toDelete;
    }
}
```

4. 実践応用

4.1 TodoListアプリでの完全なCRUD実装

```
class TodoApp {
    constructor() {
        this.crud = new AdvancedTodoCRUD();
        this.setupEventListeners();
        this.loadTasks();
    }

    setupEventListeners() {
        // Create: 新規タスク追加
        document.getElementById('addTaskForm').addEventListener('submit', (e) => {
            e.preventDefault();
            this.handleCreateTask();
        });

        // Update: 完了状態の切り替え
        document.addEventListener('change', (e) => {
            if (e.target.classList.contains('task-checkbox')) {
                this.handleToggleTask(e.target.dataset.id);
            }
        });
```

```
        // Update: インライン編集
        document.addEventListener('blur', (e) => {
            if (e.target.classList.contains('task-text-editable')) {
                this.handleUpdateTaskText(e.target.dataset.id, e.target.textContent);
            }
        }, true);

        // Delete: タスク削除
        document.addEventListener('click', (e) => {
            if (e.target.classList.contains('delete-btn')) {
                this.handleDeleteTask(e.target.dataset.id);
            }
        });

        // Read: フィルタリング
        document.getElementById('filterSelect').addEventListener('change', (e) => {
            this.handleFilterTasks(e.target.value);
        });
    }

    // Create操作の実装
    async handleCreateTask() {
        try {
            const taskInput = document.getElementById('taskInput');
            const prioritySelect = document.getElementById('prioritySelect');

            const newTask = this.crud.createTask(
                taskInput.value,
                prioritySelect.value
            );

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

            // フォームのリセット
            taskInput.value = '';
            prioritySelect.value = 'normal';

        } catch (error) {
            this.showNotification(error.message, 'error');
        }
    }

    // Read操作の実装
    loadTasks(filter = null) {
        try {
            const tasks = this.crud.readTasks(filter);
            this.renderTaskList(tasks);
        } catch (error) {
            this.showNotification('タスクの読み込みに失敗しました', 'error');
        }
    }

    // Update操作の実装
```

```
async handleToggleTask(id) {
    try {
        const updatedTask = this.crud.toggleTaskCompletion(id);
        this.updateTaskElement(id, updatedTask);

        const status = updatedTask.completed ? '完了' : '未完了';
        this.showNotification(`タスクを${status}に変更しました`, 'success');

    } catch (error) {
        this.showNotification(error.message, 'error');
    }
}

async handleUpdateTaskText(id, newText) {
    try {
        const updatedTask = this.crud.updateTask(id, { text: newText });
        this.showNotification('タスクが更新されました', 'success');

    } catch (error) {
        this.showNotification(error.message, 'error');
        // エラー時は元のテキストに戻す
        this.revertTaskText(id);
    }
}

// Delete操作の実装
async handleDeleteTask(id) {
    try {
        if (confirm('このタスクを削除しますか？')) {
            const deletedTask = this.crud.deleteTask(id);
            this.removeTaskElement(id);
            this.showNotification('タスクが削除されました', 'success');
        }
    } catch (error) {
        this.showNotification(error.message, 'error');
    }
}

// フィルタリング機能
handleFilterTasks(filterType) {
    let filter = null;

    switch (filterType) {
        case 'completed':
            filter = { type: 'completed', value: true };
            break;
        case 'pending':
            filter = { type: 'completed', value: false };
            break;
        case 'high-priority':
            filter = { type: 'priority', value: 'high' };
            break;
    }

    this.loadTasks(filter);
```

```
    }

    // UI更新メソッド
    renderTaskList(tasks) {
        const container = document.getElementById('taskList');
        container.innerHTML = '';

        tasks.forEach(task => {
            if (!task.deleted) {  // 論理削除されたタスクは表示しない
                this.renderNewTask(task);
            }
        });
    }

    renderNewTask(task) {
        const taskElement = document.createElement('div');
        taskElement.className = `task-item priority-${task.priority} ${task.completed ? 'completed' : ''}`;
        taskElement.dataset.id = task.id;

        taskElement.innerHTML = `
            <input type="checkbox" class="task-checkbox" data-id="${task.id}" ${task.completed ? 'checked' : ''}>
            <span class="task-text-editable" data-id="${task.id}" contenteditable="true">${task.text}</span>
            <span class="task-priority">${task.priority}</span>
            <button class="delete-btn" data-id="${task.id}">削除</button>
            <span class="task-date">${new Date(task.createdAt).toLocaleDateString()}</span>
        `;

        document.getElementById('taskList').appendChild(taskElement);
    }

    updateTaskElement(id, task) {
        const element = document.querySelector(`[data-id="${id}"]`).closest('.task-item');
        element.className = `task-item priority-${task.priority} ${task.completed ? 'completed' : ''}`;
    }

    removeTaskElement(id) {
        const element = document.querySelector(`[data-id="${id}"]`).closest('.task-item');
        element.remove();
    }

    showNotification(message, type) {
        // 通知の表示実装
        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', () => {
    const app = new TodoApp();
});
```

4.2 エラーハンドリングとユーザーフィードバック

```
class RobustTodoCRUD extends TodoCRUD {
    async safeOperation(operation, ...args) {
        try {
            const result = await operation.apply(this, args);
            return { success: true, data: result };
        } catch (error) {
            console.error('CRUD操作エラー:', error);
            return {
                success: false,
                error: error.message,
                code: error.code || 'UNKNOWN_ERROR'
            };
        }
    }

    // 安全なCreate操作
    async safeCreateTask(taskText, priority) {
        return this.safeOperation(this.createTask, taskText, priority);
    }

    // 安全なUpdate操作
    async safeUpdateTask(id, updates) {
        return this.safeOperation(this.updateTask, id, updates);
    }

    // 安全なDelete操作
    async safeDeleteTask(id) {
        return this.safeOperation(this.deleteTask, id);
    }
}
```

このCRUDパターンの実装により、**データの整合性を保ちながら**、**ユーザーフレンドリーな操作**を提供する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 try-catch詳解、フロントエンド状態管理 をご覧ください。