補足資料: フロントエンド 状態管理

基本(1分) - フロントエンド状態管理とは

概要: フロントエンド状態管理とは、Webアプリケーションの「現在の状態」（データ、UI設定、ユーザー操作状況など）を一貫して管理し、それに基づいてユーザーインターフェースを正しく表示・更新する仕組みです。適切な状態管理により、バグの少ない、予測可能なアプリケーションを構築できます。

なぜ状態管理が重要か:

- **データとUIの整合性**: アプリケーション内のデータとユーザーが見ている画面が常に同期している状態を保てます。
- **予測可能な動作**: 状態の変更ルールが明確であれば、アプリケーションの動作が予測しやすくなります。
- **デバッグの容易さ**: 問題が発生した時に、状態を確認することで原因を特定しやすくなります。
- **機能拡張の安全性**: 新機能追加時に既存機能に影響を与えにくくなります。

基本概念:

- **状態（State）**: アプリケーションが持つデータや設定の現在の値
- **状態更新**: 状態を変更すること（例: タスクの追加、完了状態の変更）
- **状態とUIの同期**: 状態が変更された時にUIを更新すること

基本的なパターン:

```
// 状態の定義
let state = {
  tasks: [],
  filter: 'all', // 'all', 'active', 'completed'
  isLoading: false
};

// 状態の更新
function updateState(newState) {
  state = { ...state, ...newState };
  renderUI(); // 状態変更後にUIを更新
}

// UIの描画
function renderUI() {
  // 状態に基づいてUIを構築
  console.log('現在のタスク数:', state.tasks.length);
```

```
  console.log('現在のフィルター:', state.filter);
}
```

シンプルな例:

```
// ToDoアプリケーションの基本的な状態管理
const todoState = {
  tasks: [
    { id: 1, text: '牛乳を買う', completed: false },
    { id: 2, text: '資料を作成', completed: true }
  ],
  newTaskText: '',
  showCompleted: true
};

// タスクを追加する関数
function addTask(text) {
  const newTask = {
    id: Date.now(),
    text: text,
    completed: false
  };
  todoState.tasks.push(newTask);
  todoState.newTaskText = ''; // 入力欄をクリア

  // UIを更新
  updateTaskList();
  updateInputField();
}

// UIを更新する関数
function updateTaskList() {
  const taskList = document.getElementById('taskList');
  taskList.innerHTML = todoState.tasks
    .filter(task => todoState.showCompleted || !task.completed)
    .map(task => `<li>${task.text} (${task.completed ? '完了' : '未完了'})</li>`)
    .join('');
}
```

詳細(3分) - フロントエンド状態管理の仕組みと実装方法

状態の種類と分類

1. アプリケーション状態

アプリケーション全体で共有される状態：

```
const appState = {
  // データ状態
  tasks: [],
  users: [],

  // UI状態
  currentView: 'list', // 'list', 'edit', 'settings'
  isLoading: false,
  selectedTaskId: null,

  // 設定状態
  theme: 'light',
  language: 'ja',
  itemsPerPage: 10
};
```

2. コンポーネント状態

特定の部分（コンポーネント）でのみ使用される状態：

```
// タスク入力フォームの状態
const taskFormState = {
  inputValue: '',
  isValid: true,
  validationMessage: '',
  isSubmitting: false
};

// タスク編集モーダルの状態
const editModalState = {
  isOpen: false,
  editingTaskId: null,
  tempText: '',
  hasChanges: false
};
```

3. 一時的な状態

ユーザーの操作中にのみ存在する状態：

```
// ドラッグ&ドロップの状態
const dragState = {
  isDragging: false,
  draggedTaskId: null,
  dropZone: null
```

```
};

// 検索・フィルターの状態
const filterState = {
  searchQuery: '',
  selectedTags: [],
  sortBy: 'createdAt',
  sortOrder: 'desc'
};
```

状態管理パターン

1. 単純なオブジェクト管理

```
class SimpleTodoState {
  constructor() {
    this.data = {
      tasks: [],
      filter: 'all',
      nextId: 1
    };
  }

  // 状態の取得
  getState() {
    return { ...this.data }; // 不変性を保つためコピーを返す
  }

  // タスクの追加
  addTask(text) {
    const newTask = {
      id: this.data.nextId++,
      text: text,
      completed: false,
      createdAt: new Date().toISOString()
    };
    this.data.tasks.push(newTask);
    return newTask;
  }

  // タスクの更新
  updateTask(id, updates) {
    const taskIndex = this.data.tasks.findIndex(task => task.id === id);
    if (taskIndex > -1) {
      this.data.tasks[taskIndex] = { ...this.data.tasks[taskIndex], ...updates };
      return this.data.tasks[taskIndex];
    }
    return null;
  }

  // タスクの削除
```

```
  deleteTask(id) {
    const taskIndex = this.data.tasks.findIndex(task => task.id === id);
    if (taskIndex > -1) {
      return this.data.tasks.splice(taskIndex, 1)[0];
    }
    return null;
  }

  // フィルターの設定
  setFilter(filter) {
    this.data.filter = filter;
  }

  // フィルター済みタスクの取得
  getFilteredTasks() {
    switch (this.data.filter) {
      case 'active':
        return this.data.tasks.filter(task => !task.completed);
      case 'completed':
        return this.data.tasks.filter(task => task.completed);
      default:
        return this.data.tasks;
    }
  }
}

// 使用例
const todoState = new SimpleTodoState();
todoState.addTask('牛乳を買う');
todoState.addTask('資料を作成');
console.log(todoState.getFilteredTasks());
```

2. イベント駆動型状態管理

```
class EventDrivenTodoState extends EventTarget {
  constructor() {
    super();
    this.data = {
      tasks: [],
      filter: 'all'
    };
  }

  // 状態変更時にイベントを発火
  _notifyChange(eventType, data = {}) {
    this.dispatchEvent(new CustomEvent(eventType, {
      detail: { state: this.getState(), ...data }
    }));
  }

  addTask(text) {
    const newTask = {
```

```
      id: Date.now(),
      text: text,
      completed: false,
      createdAt: new Date().toISOString()
    };
    this.data.tasks.push(newTask);
    this._notifyChange('taskAdded', { task: newTask });
    return newTask;
  }

  toggleTask(id) {
    const task = this.data.tasks.find(t => t.id === id);
    if (task) {
      task.completed = !task.completed;
      task.updatedAt = new Date().toISOString();
      this._notifyChange('taskToggled', { task: task });
    }
    return task;
  }

  getState() {
    return JSON.parse(JSON.stringify(this.data)); // Deep copy
  }
}

// 使用例
const todoState = new EventDrivenTodoState();

// 状態変更をリッスン
todoState.addEventListener('taskAdded', (event) => {
  console.log('タスクが追加されました:', event.detail.task);
  updateUI();
  saveToLocalStorage(event.detail.state);
});

todoState.addEventListener('taskToggled', (event) => {
  console.log('タスクの完了状態が変更されました:', event.detail.task);
  updateUI();
  saveToLocalStorage(event.detail.state);
});
```

状態とUIの同期パターン

1. プッシュ型更新（状態変更時にUIを更新）

```
class TodoApp {
  constructor() {
    this.state = new EventDrivenTodoState();
    this.setupEventListeners();
    this.render();
  }
```

```
  setupEventListeners() {
    // 状態の変更をリッスンしてUIを更新
    this.state.addEventListener('taskAdded', () => this.render());
    this.state.addEventListener('taskToggled', () => this.render());
    this.state.addEventListener('taskDeleted', () => this.render());
    this.state.addEventListener('filterChanged', () => this.render());
  }

  render() {
    this.renderTaskList();
    this.renderFilterButtons();
    this.renderStatistics();
  }

  renderTaskList() {
    const taskList = document.getElementById('taskList');
    const tasks = this.state.getFilteredTasks();

    taskList.innerHTML = tasks.map(task => `
      <div class="task-item ${task.completed ? 'completed' : ''}">
        <input type="checkbox" ${task.completed ? 'checked' : ''}
               onchange="app.toggleTask(${task.id})">
        <span>${task.text}</span>
        <button onclick="app.deleteTask(${task.id})">削除</button>
      </div>
    `).join('');
  }

  // ユーザーアクション
  addTask() {
    const input = document.getElementById('taskInput');
    if (input.value.trim()) {
      this.state.addTask(input.value.trim());
      input.value = '';
    }
  }

  toggleTask(id) {
    this.state.toggleTask(id);
  }
}
```

2. 仮想DOM風の差分更新

```
class VirtualDOMTodoApp {
  constructor() {
    this.state = { tasks: [], filter: 'all' };
    this.previousVirtualDOM = null;
    this.render();
  }
```

```
// 仮想DOMの生成
createVirtualDOM() {
  const filteredTasks = this.getFilteredTasks();

  return {
    type: 'div',
    props: { className: 'todo-app' },
    children: [
      {
        type: 'div',
        props: { className: 'task-list' },
        children: filteredTasks.map(task => ({
          type: 'div',
          props: {
            className: `task-item ${task.completed ? 'completed' : ''}`,
            'data-task-id': task.id
          },
          children: [
            {
              type: 'input',
              props: {
                type: 'checkbox',
                checked: task.completed,
                onchange: () => this.toggleTask(task.id)
              }
            },
            {
              type: 'span',
              props: {},
              children: [task.text]
            }
          ]
        }))
      }
    ]
  };
}

// 差分検出と実際のDOM更新
render() {
  const newVirtualDOM = this.createVirtualDOM();

  if (this.previousVirtualDOM) {
    this.updateDOM(this.previousVirtualDOM, newVirtualDOM);
  } else {
    this.createDOM(newVirtualDOM);
  }

  this.previousVirtualDOM = newVirtualDOM;
}

// 状態更新メソッド
updateState(updater) {
  const newState = updater(this.state);
  this.state = { ...this.state, ...newState };
```

```
    this.render(); // 状態更新後に再描画
  }
}
```

localStorage との連携

```
class PersistentTodoState {
  constructor(storageKey = 'todoAppState') {
    this.storageKey = storageKey;
    this.data = this.loadFromStorage();
    this.subscribers = [];
  }

  // localStorageから状態を読み込み
  loadFromStorage() {
    try {
      const saved = localStorage.getItem(this.storageKey);
      return saved ? JSON.parse(saved) : this.getDefaultState();
    } catch (error) {
      console.error('状態の読み込みに失敗:', error);
      return this.getDefaultState();
    }
  }

  // localStorageに状態を保存
  saveToStorage() {
    try {
      localStorage.setItem(this.storageKey, JSON.stringify(this.data));
      console.log('状態を保存しました');
    } catch (error) {
      console.error('状態の保存に失敗:', error);
      // 容量不足などの場合の処理
      this.handleStorageError(error);
    }
  }

  // デフォルト状態の取得
  getDefaultState() {
    return {
      tasks: [],
      filter: 'all',
      settings: {
        theme: 'light',
        autoSave: true
      },
      lastSaved: null
    };
  }

  // 状態変更の購読
  subscribe(callback) {
    this.subscribers.push(callback);
```

```
    return () => {
      this.subscribers = this.subscribers.filter(sub => sub !== callback);
    };
  }

  // 状態変更の通知
  notify() {
    this.subscribers.forEach(callback => {
      try {
        callback(this.getState());
      } catch (error) {
        console.error('状態変更の通知でエラー:', error);
      }
    });
  }

  // 状態更新（自動保存付き）
  setState(updater) {
    const newData = typeof updater === 'function' ? updater(this.data) : updater;
    this.data = { ...this.data, ...newData, lastSaved: new Date().toISOString() };

    if (this.data.settings?.autoSave) {
      this.saveToStorage();
    }

    this.notify();
  }

  // タスク操作メソッド
  addTask(text) {
    this.setState(state => ({
      tasks: [...state.tasks, {
        id: Date.now(),
        text: text,
        completed: false,
        createdAt: new Date().toISOString()
      }]
    }));
  }

  toggleTask(id) {
    this.setState(state => ({
      tasks: state.tasks.map(task =>
        task.id === id
          ? { ...task, completed: !task.completed, updatedAt: new Date().toISOString() }
          : task
      )
    }));
  }

  getState() {
    return JSON.parse(JSON.stringify(this.data));
  }
}
```

```
// 使用例
const persistentState = new PersistentTodoState();

// 状態変更の購読
const unsubscribe = persistentState.subscribe((newState) => {
  console.log('状態が更新されました:', newState);
  updateUI(newState);
});

// タスクの追加
persistentState.addTask('新しいタスク');
```

よくある間違いと注意点

❌ 間違った例1: 状態の直接変更

```
// 悪い例：状態を直接変更
state.tasks.push(newTask); // 追跡が困難
state.tasks[0].completed = true; // 副作用が予測できない
```

❌ 間違った例2: UIと状態の不整合

```
// 悪い例：UIとは独立して状態を変更
function deleteTask(id) {
  tasks = tasks.filter(task => task.id !== id);
  // UIの更新を忘れている
}
```

❌ 間違った例3: 状態の過度な分割

```
// 悪い例：関連する状態を別々に管理
let tasks = [];
let taskCount = 0;
let completedCount = 0;
// これらは全て tasks から計算できるため、重複している
```

✅ 正しい例1: 不変性を保った状態更新

```
// 良い例：新しいオブジェクトを作成
const newState = {
  ...state,
```

```
  tasks: [...state.tasks, newTask]
};
```

✅ 正しい例2: 状態変更時の一貫したUI更新

```
function updateState(newState) {
  state = { ...state, ...newState };
  renderUI(); // 必ずUIを更新
  saveToStorage(); // 必要に応じて永続化
}
```

✅ 正しい例3: 派生状態の計算

```
// 良い例：基本状態から派生状態を計算
function getStatistics(tasks) {
  return {
    total: tasks.length,
    completed: tasks.filter(t => t.completed).length,
    remaining: tasks.filter(t => !t.completed).length
  };
}
```

深掘り（コラム） - フロントエンド状態管理と関連技術

関連技術マップ（優先度順）

🔥 重要度：高 - Observer パターン

関連性: 状態の変更を複数のUIコンポーネントに効率的に通知するための基本的なパターンです。

```
class ObservableState {
  constructor(initialState = {}) {
    this.state = initialState;
    this.observers = [];
  }

  // オブザーバーの登録
  subscribe(observer) {
```

```
    this.observers.push(observer);
    return () => {
      this.observers = this.observers.filter(obs => obs !== observer);
    };
  }

  // 状態の更新と通知
  setState(updates) {
    const prevState = { ...this.state };
    this.state = { ...this.state, ...updates };

    // 全てのオブザーバーに変更を通知
    this.observers.forEach(observer => {
      observer(this.state, prevState);
    });
  }

  getState() {
    return { ...this.state };
  }
}

// 複数のUIコンポーネントで使用
const appState = new ObservableState({
  tasks: [],
  filter: 'all',
  theme: 'light'
});

// タスクリストコンポーネント
const taskListComponent = {
  init() {
    this.unsubscribe = appState.subscribe((newState, prevState) => {
      if (newState.tasks !== prevState.tasks || newState.filter !== prevState.filter) {
        this.render();
      }
    });
  },

  render() {
    const { tasks, filter } = appState.getState();
    // タスクリストのレンダリング
    console.log('タスクリストを更新:', tasks.length);
  },

  destroy() {
    this.unsubscribe?.();
  }
};

// 統計表示コンポーネント
const statisticsComponent = {
  init() {
    this.unsubscribe = appState.subscribe((newState, prevState) => {
      if (newState.tasks !== prevState.tasks) {
```

```
        this.render();
      }
    });
  },

  render() {
    const { tasks } = appState.getState();
    const completed = tasks.filter(t => t.completed).length;
    console.log(`統計を更新: ${completed}/${tasks.length} 完了`);
  }
};

// 使用例
taskListComponent.init();
statisticsComponent.init();

appState.setState({
  tasks: [
    { id: 1, text: 'タスク1', completed: false },
    { id: 2, text: 'タスク2', completed: true }
  ]
});
```

🔥 重要度：高 - State Machine（状態機械）

関連性: 複雑なUIフローや状態遷移を明確に定義し、予期しない状態変更を防ぐために重要です。

```
class TaskStateMachine {
  constructor() {
    this.states = {
      'idle': {
        transitions: {
          'START_ADD': 'adding',
          'START_EDIT': 'editing',
          'START_DELETE': 'confirming_delete'
        }
      },
      'adding': {
        transitions: {
          'SUBMIT': 'idle',
          'CANCEL': 'idle'
        }
      },
      'editing': {
        transitions: {
          'SAVE': 'idle',
          'CANCEL': 'idle'
        }
      },
      'confirming_delete': {
        transitions: {
```

```
        'CONFIRM': 'idle',
        'CANCEL': 'idle'
      }
    }
  };

  this.currentState = 'idle';
  this.data = {
    tasks: [],
    editingTaskId: null,
    deletingTaskId: null
  };
}

transition(action, payload = {}) {
  const currentStateConfig = this.states[this.currentState];
  const nextState = currentStateConfig.transitions[action];

  if (!nextState) {
    console.warn(`無効な遷移: ${this.currentState} -> ${action}`);
    return false;
  }

  console.log(`状態遷移: ${this.currentState} -> ${nextState} (${action})`);

  // 状態遷移時の副作用
  this.executeTransitionEffects(action, payload);

  this.currentState = nextState;
  return true;
}

executeTransitionEffects(action, payload) {
  switch (action) {
    case 'START_EDIT':
      this.data.editingTaskId = payload.taskId;
      break;
    case 'START_DELETE':
      this.data.deletingTaskId = payload.taskId;
      break;
    case 'SUBMIT':
      if (this.currentState === 'adding') {
        this.data.tasks.push({
          id: Date.now(),
          text: payload.text,
          completed: false
        });
      }
      break;
    case 'SAVE':
      if (this.currentState === 'editing') {
        const task = this.data.tasks.find(t => t.id === this.data.editingTaskId);
        if (task) {
          task.text = payload.text;
        }
```

```
          this.data.editingTaskId = null;
        }
        break;
      case 'CONFIRM':
        if (this.currentState === 'confirming_delete') {
          this.data.tasks = this.data.tasks.filter(t => t.id !== this.data.deletingTaskId);
          this.data.deletingTaskId = null;
        }
        break;
      case 'CANCEL':
        this.data.editingTaskId = null;
        this.data.deletingTaskId = null;
        break;
    }
  }

  canTransition(action) {
    const currentStateConfig = this.states[this.currentState];
    return !!currentStateConfig.transitions[action];
  }

  getCurrentState() {
    return this.currentState;
  }

  getData() {
    return { ...this.data };
  }
}

// 使用例
const taskMachine = new TaskStateMachine();

// UI操作に基づく状態遷移
function startEditTask(taskId) {
  if (taskMachine.canTransition('START_EDIT')) {
    taskMachine.transition('START_EDIT', { taskId });
    showEditForm(taskId);
  } else {
    console.log('現在編集モードに入れません');
  }
}

function saveTask(text) {
  if (taskMachine.canTransition('SAVE')) {
    taskMachine.transition('SAVE', { text });
    hideEditForm();
    updateTaskList();
  }
}
```

🔶 重要度：中 - Redux風のパターン

関連性: 大規模なアプリケーションで統一的な状態管理を実現するための参考パターンです。

```
// Action Types
const ACTION_TYPES = {
  ADD_TASK: 'ADD_TASK',
  TOGGLE_TASK: 'TOGGLE_TASK',
  DELETE_TASK: 'DELETE_TASK',
  SET_FILTER: 'SET_FILTER',
  SET_LOADING: 'SET_LOADING'
};

// Action Creators
const actions = {
  addTask: (text) => ({
    type: ACTION_TYPES.ADD_TASK,
    payload: { text, id: Date.now() }
  }),

  toggleTask: (id) => ({
    type: ACTION_TYPES.TOGGLE_TASK,
    payload: { id }
  }),

  deleteTask: (id) => ({
    type: ACTION_TYPES.DELETE_TASK,
    payload: { id }
  }),

  setFilter: (filter) => ({
    type: ACTION_TYPES.SET_FILTER,
    payload: { filter }
  })
};

// Reducer
function todoReducer(state = getInitialState(), action) {
  switch (action.type) {
    case ACTION_TYPES.ADD_TASK:
      return {
        ...state,
        tasks: [...state.tasks, {
          id: action.payload.id,
          text: action.payload.text,
          completed: false,
          createdAt: new Date().toISOString()
        }]
      };

    case ACTION_TYPES.TOGGLE_TASK:
      return {
        ...state,
        tasks: state.tasks.map(task =>
```

```
          task.id === action.payload.id
            ? { ...task, completed: !task.completed }
            : task
        )
      };

    case ACTION_TYPES.DELETE_TASK:
      return {
        ...state,
        tasks: state.tasks.filter(task => task.id !== action.payload.id)
      };

    case ACTION_TYPES.SET_FILTER:
      return {
        ...state,
        filter: action.payload.filter
      };

    default:
      return state;
  }
}

// Simple Store Implementation
class SimpleStore {
  constructor(reducer, initialState) {
    this.reducer = reducer;
    this.state = initialState;
    this.listeners = [];
  }

  dispatch(action) {
    console.log('Action dispatched:', action);
    this.state = this.reducer(this.state, action);
    this.listeners.forEach(listener => listener(this.state));
  }

  getState() {
    return { ...this.state };
  }

  subscribe(listener) {
    this.listeners.push(listener);
    return () => {
      this.listeners = this.listeners.filter(l => l !== listener);
    };
  }
}

function getInitialState() {
  return {
    tasks: [],
    filter: 'all',
    isLoading: false
  };
```

```
}

// 使用例
const store = new SimpleStore(todoReducer, getInitialState());

// 状態変更の購読
store.subscribe((state) => {
  console.log('State updated:', state);
  updateUI(state);
});

// アクションの実行
store.dispatch(actions.addTask('新しいタスク'));
store.dispatch(actions.toggleTask(1));
store.dispatch(actions.setFilter('completed'));
```

🔶 重要度：中 - Reactive Programming (RxJS風)

関連性: 非同期データストリームと状態管理を組み合わせて、より宣言的なコードを書くために有用です。

```
// Simple Observable Implementation
class Observable {
  constructor(subscribe) {
    this.subscribe = subscribe;
  }

  map(fn) {
    return new Observable(observer => {
      return this.subscribe({
        next: value => observer.next(fn(value)),
        error: err => observer.error(err),
        complete: () => observer.complete()
      });
    });
  }

  filter(predicate) {
    return new Observable(observer => {
      return this.subscribe({
        next: value => predicate(value) && observer.next(value),
        error: err => observer.error(err),
        complete: () => observer.complete()
      });
    });
  }

  static fromEvent(element, eventType) {
    return new Observable(observer => {
      const handler = event => observer.next(event);
      element.addEventListener(eventType, handler);
      return () => element.removeEventListener(eventType, handler);
```

```
    });
  }
}

// Reactive State Management
class ReactiveState {
  constructor(initialState) {
    this.state = initialState;
    this.stateSubject = new Observable(observer => {
      this.observer = observer;
      observer.next(this.state);
      return () => { this.observer = null; };
    });
  }

  setState(updates) {
    this.state = { ...this.state, ...updates };
    if (this.observer) {
      this.observer.next(this.state);
    }
  }

  select(selector) {
    return this.stateSubject.map(selector);
  }

  getState() {
    return { ...this.state };
  }
}

// 使用例
const reactiveState = new ReactiveState({
  tasks: [],
  filter: 'all'
});

// 特定の状態の変更を監視
const tasks$ = reactiveState.select(state => state.tasks);
const activeTaskCount$ = tasks$.map(tasks =>
  tasks.filter(task => !task.completed).length
);

// 購読
tasks$.subscribe({
  next: tasks => {
    console.log('タスクが更新されました:', tasks.length);
    renderTaskList(tasks);
  }
});

activeTaskCount$.subscribe({
  next: count => {
    console.log('未完了タスク数:', count);
    updateBadge(count);
```

```
  }
});

// 入力イベントのストリーム
const searchInput = document.getElementById('searchInput');
const searchInput$ = Observable.fromEvent(searchInput, 'input')
  .map(event => event.target.value)
  .filter(query => query.length > 2);

searchInput$.subscribe({
  next: query => {
    console.log('検索クエリ:', query);
    // 検索処理
  }
});
```

🔹 重要度：補足 – ミドルウェアパターン

関連性: 状態更新時の副作用（ログ、保存、バリデーション）を統一的に処理するために有用です。

```
// Middleware Pattern for State Management
class StateManager {
  constructor(initialState) {
    this.state = initialState;
    this.middlewares = [];
    this.subscribers = [];
  }

  use(middleware) {
    this.middlewares.push(middleware);
  }

  setState(updates) {
    const prevState = { ...this.state };
    let nextState = { ...this.state, ...updates };

    // ミドルウェアチェーンの実行
    this.middlewares.forEach(middleware => {
      nextState = middleware(prevState, nextState, updates) || nextState;
    });

    this.state = nextState;
    this.notifySubscribers();
  }

  subscribe(callback) {
    this.subscribers.push(callback);
    return () => {
      this.subscribers = this.subscribers.filter(sub => sub !== callback);
    };
  }
```

```
  notifySubscribers() {
    this.subscribers.forEach(callback => callback(this.state));
  }

  getState() {
    return { ...this.state };
  }
}

// ミドルウェアの実装例
const loggerMiddleware = (prevState, nextState, updates) => {
  console.log('State Update:', {
    previous: prevState,
    next: nextState,
    updates: updates,
    timestamp: new Date().toISOString()
  });
};

const validationMiddleware = (prevState, nextState, updates) => {
  if (updates.tasks) {
    updates.tasks.forEach(task => {
      if (!task.text || task.text.trim() === '') {
        throw new Error('タスクの内容が空です');
      }
    });
  }
};

const persistenceMiddleware = (prevState, nextState, updates) => {
  try {
    localStorage.setItem('appState', JSON.stringify(nextState));
  } catch (error) {
    console.error('状態の保存に失敗:', error);
  }
};

// 使用例
const stateManager = new StateManager({ tasks: [], filter: 'all' });

stateManager.use(validationMiddleware);
stateManager.use(loggerMiddleware);
stateManager.use(persistenceMiddleware);

// 状態更新（全てのミドルウェアが実行される）
stateManager.setState({
  tasks: [{ id: 1, text: '新しいタスク', completed: false }]
});
```

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. **Observer パターン**: 基本的な状態変更通知
2. **State Machine**: 複雑なUIフローの管理
3. **Redux風パターン**: 大規模アプリの統一的状態管理
4. **Reactive Programming**: 非同期データストリームとの統合
5. **ミドルウェアパターン**: 状態更新時の副作用処理

実践応用 - フロントエンド状態管理の総合活用

パターン1: 包括的ToDoアプリケーション（状態管理中心設計）

使用場面: 状態管理を中核とした本格的なToDoアプリケーション

```
// 状態管理の中央集権化
class ComprehensiveTodoState {
  constructor() {
    this.state = {
      // データ状態
      tasks: [],
      nextId: 1,

      // UI状態
      currentView: 'list', // 'list', 'add', 'edit'
      editingTaskId: null,
      selectedTaskIds: [],

      // フィルター状態
      filter: 'all', // 'all', 'active', 'completed'
      searchQuery: '',
      sortBy: 'createdAt',
      sortOrder: 'desc',

      // アプリケーション状態
      isLoading: false,
      lastSaved: null,
      hasUnsavedChanges: false,

      // 設定状態
      settings: {
        theme: 'light',
        autoSave: true,
        showCompletedTasks: true,
        itemsPerPage: 20
      },

      // 一時的な状態
```

```
    dragState: {
      isDragging: false,
      draggedTaskId: null,
      dropTargetId: null
    },

    // エラー状態
    errors: [],
    notifications: []
  };

  this.observers = [];
  this.history = []; // 状態履歴 (Undo/Redo用)
  this.historyIndex = -1;

  this.loadFromStorage();
  this.setupAutoSave();
}

// オブザーバーパターンの実装
subscribe(observer) {
  this.observers.push(observer);
  return () => {
    this.observers = this.observers.filter(obs => obs !== observer);
  };
}

notifyObservers(changeType, payload = {}) {
  this.observers.forEach(observer => {
    try {
      observer({
        type: changeType,
        state: this.getState(),
        payload
      });
    } catch (error) {
      console.error('Observer notification error:', error);
    }
  });
}

// 状態更新 (履歴管理付き)
setState(updates, { recordHistory = true, notifyType = 'STATE_UPDATED' } = {}) {
  const prevState = { ...this.state };

  // 履歴の記録
  if (recordHistory && this.historyIndex < this.history.length - 1) {
    this.history = this.history.slice(0, this.historyIndex + 1);
  }
  if (recordHistory) {
    this.history.push(prevState);
    this.historyIndex++;

    // 履歴サイズの制限
    if (this.history.length > 50) {
```

```
      this.history.shift();
      this.historyIndex--;
    }
  }

  this.state = { ...this.state, ...updates };
  this.state.hasUnsavedChanges = true;
  this.state.lastModified = new Date().toISOString();

  this.notifyObservers(notifyType, { prevState, updates });
}

// Undo/Redo機能
undo() {
  if (this.historyIndex >= 0) {
    this.state = { ...this.history[this.historyIndex] };
    this.historyIndex--;
    this.notifyObservers('STATE_UNDONE');
    return true;
  }
  return false;
}

redo() {
  if (this.historyIndex < this.history.length - 1) {
    this.historyIndex++;
    this.state = { ...this.history[this.historyIndex] };
    this.notifyObservers('STATE_REDONE');
    return true;
  }
  return false;
}

// タスク操作
addTask(text, options = {}) {
  const newTask = {
    id: this.state.nextId++,
    text: text.trim(),
    completed: false,
    createdAt: new Date().toISOString(),
    priority: options.priority || 'medium',
    tags: options.tags || [],
    dueDate: options.dueDate || null
  };

  this.setState({
    tasks: [...this.state.tasks, newTask],
    currentView: 'list'
  }, { notifyType: 'TASK_ADDED' });

  this.addNotification(`タスク「${text}」を追加しました`, 'success');
  return newTask;
}

updateTask(id, updates) {
```

```
  const taskIndex = this.state.tasks.findIndex(task => task.id === id);
  if (taskIndex === -1) return null;

  const updatedTask = {
    ...this.state.tasks[taskIndex],
    ...updates,
    updatedAt: new Date().toISOString()
  };

  const newTasks = [...this.state.tasks];
  newTasks[taskIndex] = updatedTask;

  this.setState({
    tasks: newTasks,
    editingTaskId: null
  }, { notifyType: 'TASK_UPDATED' });

  return updatedTask;
}

deleteTask(id) {
  const task = this.state.tasks.find(t => t.id === id);
  if (!task) return null;

  this.setState({
    tasks: this.state.tasks.filter(task => task.id !== id),
    selectedTaskIds: this.state.selectedTaskIds.filter(taskId => taskId !== id)
  }, { notifyType: 'TASK_DELETED' });

  this.addNotification(`タスク「${task.text}」を削除しました`, 'info');
  return task;
}

toggleTask(id) {
  const task = this.state.tasks.find(t => t.id === id);
  if (!task) return null;

  const updatedTask = this.updateTask(id, { completed: !task.completed });

  if (updatedTask) {
    const status = updatedTask.completed ? '完了' : '未完了';
    this.addNotification(`タスク「${task.text}」を${status}にしました`, 'success');
  }

  return updatedTask;
}

// フィルター・ソート機能
setFilter(filter) {
  this.setState({ filter }, { notifyType: 'FILTER_CHANGED' });
}

setSearchQuery(query) {
  this.setState({ searchQuery: query }, { notifyType: 'SEARCH_CHANGED' });
}
```

```
setSorting(sortBy, sortOrder = null) {
  const newSortOrder = sortOrder ||
    (this.state.sortBy === sortBy && this.state.sortOrder === 'asc' ? 'desc' : 'asc');

  this.setState({
    sortBy,
    sortOrder: newSortOrder
  }, { notifyType: 'SORT_CHANGED' });
}

// 計算済み状態 (Selectors)
getFilteredTasks() {
  let filtered = this.state.tasks;

  // フィルター適用
  switch (this.state.filter) {
    case 'active':
      filtered = filtered.filter(task => !task.completed);
      break;
    case 'completed':
      filtered = filtered.filter(task => task.completed);
      break;
  }

  // 検索クエリ適用
  if (this.state.searchQuery) {
    const query = this.state.searchQuery.toLowerCase();
    filtered = filtered.filter(task =>
      task.text.toLowerCase().includes(query) ||
      task.tags.some(tag => tag.toLowerCase().includes(query))
    );
  }

  // ソート適用
  filtered.sort((a, b) => {
    let aValue = a[this.state.sortBy];
    let bValue = b[this.state.sortBy];

    if (this.state.sortBy.includes('At') || this.state.sortBy === 'dueDate') {
      aValue = new Date(aValue || 0);
      bValue = new Date(bValue || 0);
    }

    let comparison = 0;
    if (aValue < bValue) comparison = -1;
    if (aValue > bValue) comparison = 1;

    return this.state.sortOrder === 'desc' ? -comparison : comparison;
  });

  return filtered;
}

getStatistics() {
```

```
  const tasks = this.state.tasks;
  const completed = tasks.filter(t => t.completed).length;
  const overdue = tasks.filter(t =>
    t.dueDate && new Date(t.dueDate) < new Date() && !t.completed
  ).length;

  return {
    total: tasks.length,
    completed,
    active: tasks.length - completed,
    overdue,
    completionRate: tasks.length > 0 ? Math.round((completed / tasks.length) * 100) : 0
  };
}

// 通知システム
addNotification(message, type = 'info', duration = 3000) {
  const notification = {
    id: Date.now(),
    message,
    type,
    timestamp: new Date().toISOString()
  };

  this.setState({
    notifications: [...this.state.notifications, notification]
  }, {
    recordHistory: false,
    notifyType: 'NOTIFICATION_ADDED'
  });

  if (duration > 0) {
    setTimeout(() => {
      this.removeNotification(notification.id);
    }, duration);
  }

  return notification;
}

removeNotification(id) {
  this.setState({
    notifications: this.state.notifications.filter(n => n.id !== id)
  }, {
    recordHistory: false,
    notifyType: 'NOTIFICATION_REMOVED'
  });
}

// 永続化機能
loadFromStorage() {
  try {
    const saved = localStorage.getItem('comprehensiveTodoState');
    if (saved) {
      const savedState = JSON.parse(saved);
```

```
      this.state = { ...this.state, ...savedState, hasUnsavedChanges: false };
    }
  } catch (error) {
    console.error('状態の読み込みに失敗:', error);
    this.addNotification('データの読み込みに失敗しました', 'error');
  }
}

saveToStorage() {
  try {
    const stateToSave = {
      ...this.state,
      // 一時的な状態は保存しない
      isLoading: false,
      dragState: {
        isDragging: false,
        draggedTaskId: null,
        dropTargetId: null
      },
      notifications: [],
      errors: []
    };

    localStorage.setItem('comprehensiveTodoState', JSON.stringify(stateToSave));

    this.setState({
      hasUnsavedChanges: false,
      lastSaved: new Date().toISOString()
    }, {
      recordHistory: false,
      notifyType: 'STATE_SAVED'
    });

    return true;
  } catch (error) {
    console.error('状態の保存に失敗:', error);
    this.addNotification('データの保存に失敗しました', 'error');
    return false;
  }
}

setupAutoSave() {
  // 自動保存の設定
  this.subscribe((change) => {
    if (this.state.settings.autoSave &&
        change.type !== 'STATE_SAVED' &&
        this.state.hasUnsavedChanges) {

      clearTimeout(this.autoSaveTimer);
      this.autoSaveTimer = setTimeout(() => {
        this.saveToStorage();
      }, 2000); // 2秒後に自動保存
    }
  });
```

```
    // ページ離脱時の保存
    window.addEventListener('beforeunload', (event) => {
      if (this.state.hasUnsavedChanges) {
        this.saveToStorage();
        event.returnValue = '保存されていない変更があります。';
      }
    });
  }

  getState() {
    return { ...this.state };
  }
}

// UIコンポーネントの実装
class TodoApp {
  constructor() {
    this.state = new ComprehensiveTodoState();
    this.elements = {};
    this.unsubscribers = [];

    this.init();
  }

  init() {
    this.setupElements();
    this.setupEventListeners();
    this.setupStateSubscriptions();
    this.render();
  }

  setupElements() {
    this.elements = {
      taskList: document.getElementById('taskList'),
      addTaskForm: document.getElementById('addTaskForm'),
      taskInput: document.getElementById('taskInput'),
      filterButtons: document.querySelectorAll('.filter-btn'),
      searchInput: document.getElementById('searchInput'),
      sortSelect: document.getElementById('sortSelect'),
      statistics: document.getElementById('statistics'),
      notifications: document.getElementById('notifications'),
      saveButton: document.getElementById('saveButton'),
      undoButton: document.getElementById('undoButton'),
      redoButton: document.getElementById('redoButton')
    };
  }

  setupStateSubscriptions() {
    const unsubscribe = this.state.subscribe((change) => {
      switch (change.type) {
        case 'TASK_ADDED':
        case 'TASK_UPDATED':
        case 'TASK_DELETED':
        case 'FILTER_CHANGED':
        case 'SEARCH_CHANGED':
```

```
        case 'SORT_CHANGED':
          this.renderTaskList();
          this.renderStatistics();
          break;

        case 'NOTIFICATION_ADDED':
          this.renderNotifications();
          break;

        case 'STATE_SAVED':
          this.renderSaveStatus();
          break;

        case 'STATE_UNDONE':
        case 'STATE_REDONE':
          this.render(); // 全体を再描画
          break;
      }
    });

    this.unsubscribers.push(unsubscribe);
  }

  renderTaskList() {
    const tasks = this.state.getFilteredTasks();
    const currentState = this.state.getState();

    this.elements.taskList.innerHTML = tasks.map(task => `
      <div class="task-item ${task.completed ? 'completed' : ''} ${currentState.selectedTaskIds.includes(task.id) ? 'selected' : ''}"
           data-task-id="${task.id}">
        <input type="checkbox"
               ${task.completed ? 'checked' : ''}
               onchange="app.toggleTask(${task.id})">
        <span class="task-text" onclick="app.editTask(${task.id})">${task.text}</span>
        ${task.dueDate ? `<span class="due-date">${new Date(task.dueDate).toLocaleDateString()}</span>` : ''}
        <div class="task-actions">
          <button onclick="app.editTask(${task.id})">編集</button>
          <button onclick="app.deleteTask(${task.id})">削除</button>
        </div>
      </div>
    `).join('');
  }

  renderStatistics() {
    const stats = this.state.getStatistics();
    this.elements.statistics.innerHTML = `
      <div class="stats">
        <span>全体: ${stats.total}</span>
        <span>完了: ${stats.completed}</span>
        <span>未完了: ${stats.active}</span>
        <span>完了率: ${stats.completionRate}%</span>
        ${stats.overdue > 0 ? `<span class="overdue">期限切れ: ${stats.overdue}</span>` : ''}
      </div>
    `;
```

```
  }

  renderNotifications() {
    const notifications = this.state.getState().notifications;
    this.elements.notifications.innerHTML = notifications.map(notification => `
      <div class="notification ${notification.type}">
        ${notification.message}
        <button onclick="app.removeNotification(${notification.id})">×</button>
      </div>
    `).join('');
  }

  render() {
    this.renderTaskList();
    this.renderStatistics();
    this.renderNotifications();
    this.renderSaveStatus();
  }

  // ユーザーアクション
  addTask() {
    const text = this.elements.taskInput.value.trim();
    if (text) {
      this.state.addTask(text);
      this.elements.taskInput.value = '';
    }
  }

  toggleTask(id) {
    this.state.toggleTask(id);
  }

  deleteTask(id) {
    if (confirm('このタスクを削除しますか？')) {
      this.state.deleteTask(id);
    }
  }

  saveManually() {
    this.state.saveToStorage();
  }

  undo() {
    this.state.undo();
  }

  redo() {
    this.state.redo();
  }

  destroy() {
    this.unsubscribers.forEach(unsubscribe => unsubscribe());
  }
}
```

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

// グローバルに公開（HTML内のイベントハンドラー用）
window.app = app;
```

パターン2: マルチタブ同期状態管理

使用場面: 複数のブラウザタブ間でリアルタイムに状態を同期する

```
class MultiTabSyncState {
  constructor(storageKey) {
    this.storageKey = storageKey;
    this.state = this.loadState();
    this.observers = [];
    this.setupStorageSync();
  }

  setupStorageSync() {
    // StorageEventでタブ間同期を実現
    window.addEventListener('storage', (event) => {
      if (event.key === this.storageKey && event.newValue) {
        try {
          const newState = JSON.parse(event.newValue);
          const prevState = { ...this.state };
          this.state = newState;

          this.notifyObservers({
            type: 'EXTERNAL_STATE_CHANGE',
            state: this.state,
            prevState,
            source: 'storage'
          });
        } catch (error) {
          console.error('外部状態変更の処理に失敗:', error);
        }
      }
    });

    // Broadcast Channel API（新しいブラウザで利用可能）
    if ('BroadcastChannel' in window) {
      this.broadcastChannel = new BroadcastChannel(this.storageKey);
      this.broadcastChannel.onmessage = (event) => {
        if (event.data.type === 'STATE_SYNC') {
          this.handleRemoteStateChange(event.data.state);
        }
      };
    }
  }

  setState(updates) {
    const prevState = { ...this.state };
```

```
  this.state = { ...this.state, ...updates };

  // ローカルストレージに保存
  this.saveState();

  // 他のタブに変更を通知
  if (this.broadcastChannel) {
    this.broadcastChannel.postMessage({
      type: 'STATE_SYNC',
      state: this.state,
      timestamp: Date.now()
    });
  }

  this.notifyObservers({
    type: 'LOCAL_STATE_CHANGE',
    state: this.state,
    prevState,
    updates
  });
}

handleRemoteStateChange(newState) {
  const prevState = { ...this.state };

  // 競合解決（最新のタイムスタンプを採用）
  if (newState.lastModified > this.state.lastModified) {
    this.state = newState;

    this.notifyObservers({
      type: 'EXTERNAL_STATE_CHANGE',
      state: this.state,
      prevState,
      source: 'broadcast'
    });
  }
}

loadState() {
  try {
    const saved = localStorage.getItem(this.storageKey);
    return saved ? JSON.parse(saved) : this.getDefaultState();
  } catch (error) {
    console.error('状態の読み込みに失敗:', error);
    return this.getDefaultState();
  }
}

saveState() {
  try {
    localStorage.setItem(this.storageKey, JSON.stringify(this.state));
  } catch (error) {
    console.error('状態の保存に失敗:', error);
  }
}
```

```
  subscribe(observer) {
    this.observers.push(observer);
    return () => {
      this.observers = this.observers.filter(obs => obs !== observer);
    };
  }

  notifyObservers(change) {
    this.observers.forEach(observer => {
      try {
        observer(change);
      } catch (error) {
        console.error('Observer notification error:', error);
      }
    });
  }

  getDefaultState() {
    return {
      tasks: [],
      lastModified: Date.now(),
      tabId: Math.random().toString(36).substr(2, 9)
    };
  }

  getState() {
    return { ...this.state };
  }
}
```

技術選択の判断基準まとめ

状況A（小規模アプリケーション）の場合: 単純なオブジェクト管理とObserverパターンを使用。

状況B（中規模アプリケーション）の場合: State Machineやイベント駆動型の状態管理を導入。

状況C（大規模アプリケーション）の場合: Redux風のパターンやミドルウェアシステムを活用。

状況D（リアルタイム同期が必要）の場合: Reactive ProgrammingやBroadcast Channel APIを組み合わせる。

重要な原則:

1. **予測可能な状態遷移の確保**
2. **UIと状態の一貫した同期**
3. **適切なスコープでの状態管理**

4. **パフォーマンスとメモリ効率の考慮**

関連する補足資料

- DOM イベント処理: ユーザーインタラクションによる状態変更の実装方法。
- localStorage API詳解: 状態の永続化とブラウザ間同期の実装方法。
- JavaScript オブジェクト詳解: 状態オブジェクトの設計と操作方法。
- JSON操作詳解: 状態のシリアライゼーションと復元方法。
- try-catch詳解: 状態管理におけるエラーハンドリング。

最終更新日: 2024/06/13