補足資料: JavaScriptオブジェクト詳解

基本(1分) - JavaScriptオブジェクトとは

概要: JavaScriptのオブジェクトは、関連する複数のデータ（プロパティ）と処理（メソッド）を1つにまとめて管理できる、非常に重要なデータ構造です。
現実世界の「もの」をプログラム上で表現するのに最適で、データの構造化とコードの整理に欠かせません。

なぜオブジェクトが重要か:

- **データの構造化**: 関連する情報をひとまとまりにして管理できます。
- **可読性の向上**: データの意味や関係性が明確になり、コードが理解しやすくなります。
- **再利用性**: 同じ構造のデータを効率的に作成・操作できます。
- **現実世界のモデル化**: 人、商品、タスクなどの概念をプログラム上で自然に表現できます。

基本構文:

```
// オブジェクトリテラル記法（最も一般的）
const objectName = {
  property1: value1,
  property2: value2,
  methodName: function() {
    // メソッドの処理
  }
};
```

基本的なアクセス方法:

```
// ドット記法（プロパティ名が固定の場合）
objectName.property1

// ブラケット記法（プロパティ名が動的な場合）
objectName['property1']
objectName[variableName]
```

シンプルな例:

```
// ToDoタスクをオブジェクトで表現
const task = {
  id: 1,
  text: '牛乳を買う',
  completed: false,
  dueDate: '2024-06-15'
};

// プロパティにアクセス
console.log(task.text);       // '牛乳を買う'
console.log(task.completed);  // false
console.log(task['dueDate']); // '2024-06-15'

// プロパティを変更
task.completed = true;
console.log(task.completed);  // true
```

詳細(3分) - オブジェクトの仕組みと操作方法

オブジェクトの作成方法

1. オブジェクトリテラル記法（推奨）

```
const user = {
  name: '田中太郎',
  age: 25,
  email: 'tanaka@example.com',
  isActive: true
};
```

2. new Object()構文

```
const user = new Object();
user.name = '田中太郎';
user.age = 25;
//  （通常はリテラル記法の方が簡潔で推奨される）
```

3. Object.create()（上級者向け）

```
const userPrototype = { greet: function() { return `こんにちは、${this.name}です`; } };
const user = Object.create(userPrototype);
user.name = '田中太郎';
```

プロパティの操作

プロパティの追加

```
const task = { id: 1, text: '買い物' };

// ドット記法で追加
task.completed = false;
task.dueDate = '2024-06-15';

// ブラケット記法で追加（動的なキー名の場合）
const propertyName = 'priority';
task[propertyName] = 'high';
task['created' + 'At'] = new Date().toISOString();

console.log(task);
// { id: 1, text: '買い物', completed: false, dueDate: '2024-06-15', priority: 'high', createdAt: '...' }
```

プロパティの更新

```
const user = { name: '太郎', age: 20 };

// 既存のプロパティを更新
user.age = 21;
user['name'] = '田中太郎'; // フルネームに変更

console.log(user); // { name: '田中太郎', age: 21 }
```

プロパティの削除

```
const task = { id: 1, text: '買い物', completed: false, tempData: 'temporary' };

// delete演算子を使用
delete task.tempData;
delete task['completed']; // ブラケット記法でも可能

console.log(task); // { id: 1, text: '買い物' }
```

メソッド（関数プロパティ）

オブジェクトのプロパティとして関数を定義することで、そのオブジェクトに関連する処理をまとめることができます。

```
const calculator = {
  result: 0,

  add: function(number) {
    this.result += number;
    return this; // メソッドチェーンを可能にする
  },

  multiply: function(number) {
    this.result *= number;
    return this;
  },

  getResult: function() {
    return this.result;
  },

  reset: function() {
    this.result = 0;
    return this;
  }
};

// 使用例
const finalResult = calculator.add(5).multiply(3).getResult();
console.log(finalResult); // 15

// ES6のメソッド短縮記法
const task = {
  id: 1,
  text: '買い物',
  completed: false,

  // function キーワードを省略できる
  toggle() {
    this.completed = !this.completed;
  },

  getInfo() {
    return `${this.text} (${this.completed ? '完了' : '未完了'})`;
  }
};

task.toggle();
console.log(task.getInfo()); // '買い物 (完了)'
```

オブジェクトの反復処理

1. for...in文

```
const user = { name: '太郎', age: 25, city: '東京' };

for (const key in user) {
  console.log(`${key}: ${user[key]}`);
}
// name: 太郎
// age: 25
// city: 東京
```

2. Object.keys()（キーのみ）

```
const user = { name: '太郎', age: 25, city: '東京' };

Object.keys(user).forEach(key => {
  console.log(`キー: ${key}, 値: ${user[key]}`);
});
```

3. Object.values()（値のみ）

```
const scores = { math: 85, english: 92, science: 78 };

const totalScore = Object.values(scores).reduce((sum, score) => sum + score, 0);
console.log(`合計点: ${totalScore}`); // 合計点: 255
```

4. Object.entries()（キーと値のペア）

```
const user = { name: '太郎', age: 25, city: '東京' };

Object.entries(user).forEach(([key, value]) => {
  console.log(`${key}: ${value}`);
});

// オブジェクトを配列に変換してさらに処理
const userArray = Object.entries(user);
console.log(userArray); // [['name', '太郎'], ['age', 25], ['city', '東京']]
```

ネストしたオブジェクトの操作

```
const todoApp = {
  settings: {
    theme: 'dark',
    language: 'ja',
    itemsPerPage: 10
  },
  user: {
    id: 1,
    profile: {
      name: '太郎',
      email: 'taro@example.com'
    },
    preferences: {
      notifications: true,
      autoSave: true
    }
  },
  tasks: [
    { id: 1, text: '買い物', completed: false },
    { id: 2, text: '掃除', completed: true }
  ]
};

// ネストしたプロパティへのアクセス
console.log(todoApp.settings.theme);                    // 'dark'
console.log(todoApp.user.profile.name);                 // '太郎'
console.log(todoApp.user.preferences.notifications);    // true

// ネストしたプロパティの更新
todoApp.settings.theme = 'light';
todoApp.user.profile.name = '田中太郎';

// 安全なアクセス（プロパティが存在しない場合の対処）
const userName = todoApp.user && todoApp.user.profile && todoApp.user.profile.name;
console.log(userName); // '田中太郎' または undefined

// ES2020の Optional Chaining (?.) が利用可能な環境では
// const userName = todoApp.user?.profile?.name;
```

よくある間違いと注意点

❌ 間違った例1: null/undefinedのプロパティアクセス

```
const obj = null;
console.log(obj.property); // TypeError: Cannot read property 'property' of null

const user = {};
console.log(user.profile.name); // TypeError: Cannot read property 'name' of undefined
```

❌ 間違った例2: 予約語をプロパティ名にする際の注意

```
const obj = {
  class: 'myClass', // 'class'は予約語だが、プロパティ名としては使用可能
  var: 'variable'   // ただし、ドット記法でアクセスする際に問題になる場合がある
};

// ブラケット記法を使用する必要がある場合
console.log(obj['class']); // 'myClass'
```

❌ 間違った例3: オブジェクトの比較

```
const obj1 = { name: '太郎' };
const obj2 = { name: '太郎' };
console.log(obj1 === obj2); // false（参照が異なる）

// 正しい比較方法は文脈によるが、一例として：
console.log(obj1.name === obj2.name); // true（プロパティ値の比較）
```

✅ 正しい例1: 安全なプロパティアクセス

```
const user = {}; // 空のオブジェクト

// 事前に存在確認
if (user.profile && user.profile.name) {
  console.log(user.profile.name);
} else {
  console.log('プロフィール名が設定されていません');
}

// デフォルト値を使用
const userName = (user.profile && user.profile.name) || 'ゲスト';
console.log(userName); // 'ゲスト'
```

✅ 正しい例2: オブジェクトのコピー

```
const original = { name: '太郎', age: 25 };

// 浅いコピー（第1レベルのプロパティのみ）
const shallowCopy = { ...original }; // スプレッド演算子
// または const shallowCopy = Object.assign({}, original);

shallowCopy.name = '次郎';
```

```
console.log(original.name); // '太郎'（元のオブジェクトは変更されない）
console.log(shallowCopy.name); // '次郎'
```

✅ 正しい例3: オブジェクトの存在確認

```
const config = {
  api: {
    baseUrl: 'https://api.example.com'
  }
};

// hasOwnPropertyを使用した存在確認
if (config.hasOwnProperty('api')) {
  console.log('API設定が存在します');
}

// in演算子を使用した存在確認
if ('api' in config) {
  console.log('API設定が存在します');
}

// undefinedとの比較による確認
if (config.api !== undefined) {
  console.log('API設定が存在します');
}
```

深掘り（コラム） - JavaScriptオブジェクトと関連技術

関連技術マップ（優先度順）

🔥 重要度：高 - ES6 Class構文

関連性: オブジェクトの作成パターンを構造化し、より読みやすく再利用可能なコードを書くために重要です。

```
// クラス定義
class Task {
  constructor(id, text, dueDate = null) {
    this.id = id;
    this.text = text;
    this.completed = false;
    this.dueDate = dueDate;
    this.createdAt = new Date().toISOString();
```

```
  }

  toggle() {
    this.completed = !this.completed;
    this.updatedAt = new Date().toISOString();
  }

  getInfo() {
    const status = this.completed ? '✓ 完了' : '○ 未完了';
    const dueDateInfo = this.dueDate ? ` (期限: ${this.dueDate})` : '';
    return `${status} ${this.text}${dueDateInfo}`;
  }

  static fromJson(jsonString) {
    const data = JSON.parse(jsonString);
    return new Task(data.id, data.text, data.dueDate);
  }
}

// クラスの使用
const task1 = new Task(1, '買い物', '2024-06-15');
const task2 = new Task(2, '掃除');

console.log(task1.getInfo()); // '○ 未完了 買い物 (期限: 2024-06-15)'
task1.toggle();
console.log(task1.getInfo()); // '✓ 完了 買い物 (期限: 2024-06-15)'

// 静的メソッドの使用
const task3 = Task.fromJson('{"id": 3, "text": "読書", "dueDate": "2024-06-20"}');
```

🔥 重要度：高 - 分割代入（Destructuring）

関連性: オブジェクトのプロパティを効率的に抽出し、変数に代入するために欠かせません。

```
const user = {
  id: 1,
  name: '田中太郎',
  email: 'tanaka@example.com',
  profile: {
    age: 25,
    city: '東京'
  }
};

// 基本的な分割代入
const { name, email } = user;
console.log(name);  // '田中太郎'
console.log(email); // 'tanaka@example.com'

// 変数名を変更する場合
const { name: userName, email: userEmail } = user;
```

```
console.log(userName);  // '田中太郎'
console.log(userEmail); // 'tanaka@example.com'

// デフォルト値を設定
const { name, email, phone = '未設定' } = user;
console.log(phone); // '未設定'

// ネストしたオブジェクトの分割代入
const { profile: { age, city } } = user;
console.log(age);  // 25
console.log(city); // '東京'

// 関数の引数での分割代入
function displayUserInfo({ name, email, profile: { age } }) {
  console.log(`名前: ${name}, メール: ${email}, 年齢: ${age}`);
}
displayUserInfo(user); // '名前: 田中太郎, メール: tanaka@example.com, 年齢: 25'

// 配列との組み合わせ
const tasks = [
  { id: 1, text: '買い物', completed: false },
  { id: 2, text: '掃除', completed: true }
];

tasks.forEach(({ id, text, completed }) => {
  console.log(`[${id}] ${text}: ${completed ? '完了' : '未完了'}`);
});
```

🔥 重要度：高 - スプレッド演算子（...）

関連性: オブジェクトのコピー、マージ、プロパティの動的追加に非常に有用です。

```
const baseTask = {
  id: 1,
  text: '買い物',
  completed: false
};

// オブジェクトのコピー
const taskCopy = { ...baseTask };
taskCopy.completed = true;
console.log(baseTask.completed); // false（元のオブジェクトは変更されない）
console.log(taskCopy.completed); // true

// オブジェクトのマージ
const additionalInfo = {
  dueDate: '2024-06-15',
  priority: 'high'
};
const completedTask = { ...baseTask, ...additionalInfo };
console.log(completedTask);
```

```
// { id: 1, text: '買い物', completed: false, dueDate: '2024-06-15', priority: 'high' }

// プロパティの上書き
const updatedTask = { ...baseTask, completed: true, updatedAt: new Date().toISOString() };
console.log(updatedTask.completed); // true

// 関数の引数での活用
function updateTask(originalTask, updates) {
  return { ...originalTask, ...updates, updatedAt: new Date().toISOString() };
}

const newTask = updateTask(baseTask, { completed: true, priority: 'low' });
console.log(newTask);
```

🔶 重要度：中 - Objectユーティリティメソッド

関連性: オブジェクトの操作、制御、変換に便利なメソッド群です。

```
const user = { name: '太郎', age: 25, city: '東京' };

// Object.assign() - オブジェクトの浅いコピーとマージ
const userCopy = Object.assign({}, user);
const extendedUser = Object.assign({}, user, { email: 'taro@example.com', phone: '090-1234-5678' });

// Object.freeze() - オブジェクトの変更を禁止
const frozenUser = Object.freeze({ ...user });
// frozenUser.name = '次郎'; // strict mode では TypeError が発生

// Object.seal() - プロパティの追加・削除を禁止（値の変更は可能）
const sealedUser = Object.seal({ ...user });
sealedUser.age = 26; // 変更可能
// sealedUser.email = 'test@example.com'; // 追加不可

// Object.defineProperty() - プロパティの詳細制御
const task = {};
Object.defineProperty(task, 'id', {
  value: 1,
  writable: false,   // 変更不可
  enumerable: true,  // for...in で列挙される
  configurable: false // プロパティ削除不可
});

// Object.getOwnPropertyDescriptor() - プロパティの設定情報を取得
const descriptor = Object.getOwnPropertyDescriptor(task, 'id');
console.log(descriptor);
// { value: 1, writable: false, enumerable: true, configurable: false }
```

🔶 重要度：中 - プロトタイプチェーン基本

関連性: JavaScriptのオブジェクト継承メカニズムの理解に重要です。

```
// コンストラクター関数
function Task(id, text) {
  this.id = id;
  this.text = text;
  this.completed = false;
}

// プロトタイプにメソッドを追加
Task.prototype.toggle = function() {
  this.completed = !this.completed;
};

Task.prototype.getInfo = function() {
  return `${this.text}: ${this.completed ? '完了' : '未完了'}`;
};

// インスタンス作成
const task1 = new Task(1, '買い物');
const task2 = new Task(2, '掃除');

console.log(task1.getInfo()); // '買い物: 未完了'
task1.toggle();
console.log(task1.getInfo()); // '買い物: 完了'

// プロトタイプチェーンの確認
console.log(task1.hasOwnProperty('id'));       // true（自分のプロパティ）
console.log(task1.hasOwnProperty('toggle'));   // false（プロトタイプのメソッド）
console.log('toggle' in task1);                // true（プロトタイプチェーンで見つかる）
```

🔹 重要度：補足 - JSON操作との連携

関連性: オブジェクトとJSONの相互変換は、データの永続化や通信で必須です。

```
const task = {
  id: 1,
  text: '買い物',
  completed: false,
  dueDate: new Date('2024-06-15'),
  tags: ['日用品', '食料']
};

// オブジェクト → JSON文字列
const jsonString = JSON.stringify(task);
console.log(jsonString);

// JSON文字列 → オブジェクト
const parsedTask = JSON.parse(jsonString);
console.log(parsedTask);
```

```
// 注意: Date オブジェクトは文字列になる
console.log(typeof task.dueDate);        // 'object' (Date)
console.log(typeof parsedTask.dueDate);  // 'string'

// カスタムシリアライザを使用
const taskWithReplacer = JSON.stringify(task, (key, value) => {
  if (value instanceof Date) {
    return { __type: 'Date', value: value.toISOString() };
  }
  return value;
});

const taskWithReviver = JSON.parse(taskWithReplacer, (key, value) => {
  if (value && value.__type === 'Date') {
    return new Date(value.value);
  }
  return value;
});

console.log(taskWithReviver.dueDate instanceof Date); // true
```

技術の全体像

これらの技術は、以下のように組み合わせて使用されます：

1. **基本的なオブジェクト**: データの構造化
2. **Class構文**: オブジェクトの構造化とメソッドの整理
3. **分割代入**: 効率的なプロパティ抽出
4. **スプレッド演算子**: オブジェクトの複製・マージ
5. **Objectユーティリティ**: 高度なオブジェクト制御
6. **JSON連携**: データの永続化・通信

実践応用 - オブジェクトの総合活用

パターン1: ToDoアプリでのオブジェクト設計

使用場面: ToDoリストアプリケーションでの包括的なオブジェクト活用

```
// タスク管理クラス
class Task {
  constructor(text, options = {}) {
    this.id = options.id || Date.now();
    this.text = text;
```

```
    this.completed = options.completed || false;
    this.dueDate = options.dueDate || null;
    this.priority = options.priority || 'medium';
    this.tags = options.tags || [];
    this.createdAt = options.createdAt || new Date().toISOString();
    this.updatedAt = options.updatedAt || new Date().toISOString();
  }

  toggle() {
    this.completed = !this.completed;
    this.updatedAt = new Date().toISOString();
    return this;
  }

  updateText(newText) {
    this.text = newText;
    this.updatedAt = new Date().toISOString();
    return this;
  }

  addTag(tag) {
    if (!this.tags.includes(tag)) {
      this.tags.push(tag);
      this.updatedAt = new Date().toISOString();
    }
    return this;
  }

  toJSON() {
    return {
      id: this.id,
      text: this.text,
      completed: this.completed,
      dueDate: this.dueDate,
      priority: this.priority,
      tags: [...this.tags],
      createdAt: this.createdAt,
      updatedAt: this.updatedAt
    };
  }

  static fromJSON(jsonData) {
    return new Task(jsonData.text, jsonData);
  }
}

// アプリケーション設定管理
class AppSettings {
  constructor() {
    this.config = {
      theme: 'light',
      language: 'ja',
      itemsPerPage: 10,
      sortBy: 'createdAt',
      sortOrder: 'desc',
```

```
      showCompletedTasks: true,
      notifications: {
        enabled: true,
        dueDateReminder: true,
        completionSound: false
      }
    };
  }

  get(path) {
    return path.split('.').reduce((obj, key) => obj && obj[key], this.config);
  }

  set(path, value) {
    const keys = path.split('.');
    const lastKey = keys.pop();
    const target = keys.reduce((obj, key) => obj[key] = obj[key] || {}, this.config);
    target[lastKey] = value;
  }

  toJSON() {
    return JSON.stringify(this.config);
  }

  fromJSON(jsonString) {
    try {
      this.config = JSON.parse(jsonString);
    } catch (error) {
      console.error('設定の読み込みに失敗:', error.message);
    }
  }
}

// ToDoアプリケーションメインクラス
class TodoApp {
  constructor() {
    this.tasks = [];
    this.settings = new AppSettings();
    this.filters = {
      status: 'all',     // 'all', 'active', 'completed'
      priority: 'all',   // 'all', 'high', 'medium', 'low'
      tags: []
    };
  }

  addTask(text, options = {}) {
    const task = new Task(text, options);
    this.tasks.push(task);
    return task;
  }

  getTask(id) {
    return this.tasks.find(task => task.id === id);
  }
```

```
updateTask(id, updates) {
  const task = this.getTask(id);
  if (task) {
    Object.assign(task, updates);
    task.updatedAt = new Date().toISOString();
  }
  return task;
}

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

getFilteredTasks() {
  return this.tasks.filter(task => {
    // ステータスフィルター
    if (this.filters.status === 'active' && task.completed) return false;
    if (this.filters.status === 'completed' && !task.completed) return false;

    // 優先度フィルター
    if (this.filters.priority !== 'all' && task.priority !== this.filters.priority) return false;

    // タグフィルター
    if (this.filters.tags.length > 0) {
      const hasMatchingTag = this.filters.tags.some(tag => task.tags.includes(tag));
      if (!hasMatchingTag) return false;
    }

    return true;
  });
}

getSortedTasks(tasks = this.getFilteredTasks()) {
  const sortBy = this.settings.get('sortBy');
  const sortOrder = this.settings.get('sortOrder');

  return [...tasks].sort((a, b) => {
    let aValue = a[sortBy];
    let bValue = b[sortBy];

    // 日付文字列の場合は Date オブジェクトに変換
    if (sortBy.includes('At') || 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 sortOrder === 'desc' ? -comparison : comparison;
```

```
    });
  }

  getStatistics() {
    const total = this.tasks.length;
    const completed = this.tasks.filter(task => task.completed).length;
    const active = total - completed;

    // 優先度別統計
    const byPriority = this.tasks.reduce((stats, task) => {
      stats[task.priority] = (stats[task.priority] || 0) + 1;
      return stats;
    }, {});

    // タグ別統計
    const byTags = this.tasks.reduce((stats, task) => {
      task.tags.forEach(tag => {
        stats[tag] = (stats[tag] || 0) + 1;
      });
      return stats;
    }, {});

    return {
      total,
      completed,
      active,
      completionRate: total > 0 ? Math.round((completed / total) * 100) : 0,
      byPriority,
      byTags
    };
  }

  exportToJSON() {
    return JSON.stringify({
      tasks: this.tasks.map(task => task.toJSON()),
      settings: this.settings.config,
      filters: this.filters,
      exportedAt: new Date().toISOString()
    }, null, 2);
  }

  importFromJSON(jsonString) {
    try {
      const data = JSON.parse(jsonString);

      this.tasks = data.tasks.map(taskData => Task.fromJSON(taskData));
      this.settings.config = { ...this.settings.config, ...data.settings };
      this.filters = { ...this.filters, ...data.filters };

      console.log(`${this.tasks.length}件のタスクをインポートしました`);
    } catch (error) {
      console.error('インポートに失敗:', error.message);
    }
  }
}
```

```
// 使用例
const app = new TodoApp();

// タスクの追加
app.addTask('牛乳を買う', {
  priority: 'high',
  dueDate: '2024-06-15',
  tags: ['買い物', '食料']
});

app.addTask('部屋を掃除する', {
  priority: 'medium',
  tags: ['家事']
});

app.addTask('レポートを書く', {
  priority: 'high',
  dueDate: '2024-06-20',
  tags: ['仕事', '締切']
});

// フィルターの設定
app.filters.priority = 'high';
app.filters.tags = ['買い物'];

// フィルター・ソート済みタスクの取得
const filteredTasks = app.getSortedTasks();
console.log('高優先度の買い物タスク:', filteredTasks);

// 統計情報の取得
const stats = app.getStatistics();
console.log('統計情報:', stats);

// データのエクスポート・インポート
const exportedData = app.exportToJSON();
console.log('エクスポートされたデータ:', exportedData);
```

パターン2: localStorage + オブジェクト + JSON の組み合わせ

使用場面: オブジェクトデータの永続化とローカルストレージ連携

```
class DataManager {
  constructor(storageKey) {
    this.storageKey = storageKey;
  }

  save(data) {
    try {
      const jsonString = JSON.stringify(data, this.replacer);
      localStorage.setItem(this.storageKey, jsonString);
      return true;
```

```
    } catch (error) {
      console.error('データ保存エラー:', error.message);
      return false;
    }
  }

  load(defaultValue = null) {
    try {
      const jsonString = localStorage.getItem(this.storageKey);
      if (jsonString === null) return defaultValue;
      return JSON.parse(jsonString, this.reviver);
    } catch (error) {
      console.error('データ読み込みエラー:', error.message);
      return defaultValue;
    }
  }

  // Date オブジェクトを適切にシリアライズ
  replacer(key, value) {
    if (value instanceof Date) {
      return { __type: 'Date', value: value.toISOString() };
    }
    return value;
  }

  // Date オブジェクトを適切にデシリアライズ
  reviver(key, value) {
    if (value && value.__type === 'Date') {
      return new Date(value.value);
    }
    return value;
  }

  remove() {
    localStorage.removeItem(this.storageKey);
  }

  exists() {
    return localStorage.getItem(this.storageKey) !== null;
  }
}

// TodoApp との連携
class PersistentTodoApp extends TodoApp {
  constructor() {
    super();
    this.dataManager = new DataManager('todoApp_data');
    this.loadFromStorage();
  }

  loadFromStorage() {
    const savedData = this.dataManager.load();
    if (savedData) {
      if (savedData.tasks) {
        this.tasks = savedData.tasks.map(taskData => Task.fromJSON(taskData));
```

```
      }
      if (savedData.settings) {
        this.settings.config = { ...this.settings.config, ...savedData.settings };
      }
      if (savedData.filters) {
        this.filters = { ...this.filters, ...savedData.filters };
      }
      console.log(`${this.tasks.length}件のタスクを復元しました`);
    }
  }

  saveToStorage() {
    const dataToSave = {
      tasks: this.tasks.map(task => task.toJSON()),
      settings: this.settings.config,
      filters: this.filters,
      lastSaved: new Date().toISOString()
    };

    if (this.dataManager.save(dataToSave)) {
      console.log('データを保存しました');
    } else {
      console.error('データの保存に失敗しました');
    }
  }

  // 元のメソッドをオーバーライドして自動保存機能を追加
  addTask(text, options = {}) {
    const task = super.addTask(text, options);
    this.saveToStorage();
    return task;
  }

  updateTask(id, updates) {
    const task = super.updateTask(id, updates);
    if (task) {
      this.saveToStorage();
    }
    return task;
  }

  deleteTask(id) {
    const task = super.deleteTask(id);
    if (task) {
      this.saveToStorage();
    }
    return task;
  }
}

// 使用例
const persistentApp = new PersistentTodoApp();

// データは自動的に localStorage に保存される
persistentApp.addTask('永続化テスト', { priority: 'high' });
```

パターン3: オブジェクト指向設計パターンの活用

使用場面: 大規模なアプリケーションでの設計パターン適用

```
// Observer パターンの実装
class EventEmitter {
  constructor() {
    this.events = {};
  }

  on(eventName, callback) {
    if (!this.events[eventName]) {
      this.events[eventName] = [];
    }
    this.events[eventName].push(callback);
  }

  emit(eventName, data) {
    if (this.events[eventName]) {
      this.events[eventName].forEach(callback => callback(data));
    }
  }

  off(eventName, callback) {
    if (this.events[eventName]) {
      this.events[eventName] = this.events[eventName].filter(cb => cb !== callback);
    }
  }
}

// イベント駆動型 TodoApp
class EventDrivenTodoApp extends EventEmitter {
  constructor() {
    super();
    this.tasks = [];
    this.setupEventListeners();
  }

  setupEventListeners() {
    this.on('taskAdded', (task) => {
      console.log(`タスク追加: ${task.text}`);
      this.saveToStorage();
    });

    this.on('taskCompleted', (task) => {
      console.log(`タスク完了: ${task.text}`);
      this.checkAllTasksCompleted();
    });

    this.on('allTasksCompleted', () => {
      console.log('🎉 すべてのタスクが完了しました！');
```

```
    });
  }

  addTask(text, options = {}) {
    const task = new Task(text, options);
    this.tasks.push(task);
    this.emit('taskAdded', task);
    return task;
  }

  completeTask(id) {
    const task = this.tasks.find(t => t.id === id);
    if (task && !task.completed) {
      task.toggle();
      this.emit('taskCompleted', task);
    }
    return task;
  }

  checkAllTasksCompleted() {
    const allCompleted = this.tasks.length > 0 &&
                         this.tasks.every(task => task.completed);
    if (allCompleted) {
      this.emit('allTasksCompleted');
    }
  }

  saveToStorage() {
    // 保存処理（簡略化）
    console.log('データを保存中...');
  }
}

// 使用例
const eventApp = new EventDrivenTodoApp();

eventApp.addTask('タスク1');
eventApp.addTask('タスク2');

// すべてのタスクを完了させる
eventApp.tasks.forEach(task => eventApp.completeTask(task.id));
```

技術選択の判断基準まとめ

状況A（シンプルなデータ管理）の場合: 基本的なオブジェクトリテラルと分割代入、スプレッド演算子を使用。

状況B（構造化されたデータ管理）の場合: ES6 Class構文を使用し、メソッドによるデータ操作を整理。

状況C（複雑なアプリケーション）の場合: 設計パターン（Observer、Factory等）を組み合わせ、イベント駆動やコンポーネント化を適用。

重要な原則:

1. **データの関連性に基づいてオブジェクトを設計する**
2. **不変性とコピーを適切に使い分ける**
3. **型安全性とエラーハンドリングを考慮する**
4. **パフォーマンスと可読性のバランスを取る**

関連する補足資料

- JSON操作詳解: オブジェクトとJSONの相互変換、シリアライゼーションの詳細。
- localStorage API詳解: オブジェクトデータの永続化方法。
- データ構造設計: アプリケーションに適したオブジェクト構造の設計方法。
- try-catch詳解: オブジェクト操作時のエラーハンドリング。
- ブラウザストレージ比較: 各種ストレージでのオブジェクト管理。

最終更新日: 2024/06/13