補足資料2: JSON操作詳解

基本(1分) - JSONとは

概要: JSON (JavaScript Object Notation) は、データを文字列として保存・送信するための軽量な形式で、JavaScriptオブジェクトと文字列を相互変換できます。

基本構文:

```
// 基本的な変換操作
const オブジェクト = {name: "太郎", age: 25};
const JSON文字列 = JSON.stringify(オブジェクト);  // オブジェクト → 文字列
const復元オブジェクト = JSON.parse(JSON文字列);    // 文字列 → オブジェクト
```

シンプルな例:

```
// 最もシンプルな使用例
const userData = {name: "田中", score: 100};
const saved = JSON.stringify(userData);  // '{"name":"田中","score":100}'
const restored = JSON.parse(saved);      // {name: "田中", score: 100}
```

詳細(3分) - JSONの仕組みと使い方

動作の仕組み

JSON.stringify()はJavaScriptの値をJSON文字列に変換し、JSON.parse()はJSON文字列をJavaScriptの値に復元します。localStorageなど文字列しか保存できない場所で、オブジェクトや配列を扱うために必須の技術です。

実行フロー

1. **stringify時**: JavaScript値 → 内部パース処理 → JSON文字列
2. **parse時**: JSON文字列 → 構文解析 → JavaScript値
3. **エラー時**: 不正な文字列 → SyntaxError例外

例:

```
// より詳細な例（配列とネストオブジェクト）
const complexData = {
    tasks: [
        {id: 1, text: "買い物", completed: true},
        {id: 2, text: "掃除", completed: false}
    ],
    settings: {
        theme: "dark",
        notifications: true
    },
    lastUpdated: "2025-06-13T10:30:00Z"
};

// 複雑なデータもそのまま変換可能
const jsonString = JSON.stringify(complexData);
console.log(jsonString);
// '{"tasks":[{"id":1,"text":"買い物","completed":true}...]...}'

// 完全に復元される
const restored = JSON.parse(jsonString);
console.log(restored.tasks[0].text); // "買い物"
```

よくある間違いと注意点

❌ 間違った例:

```
// 不正なJSON文字列をparseしようとする
const broken = '{"name": "太郎", "age": 25,}'; // 末尾のカンマが不正
const result = JSON.parse(broken); // SyntaxError が発生

// null チェックなしでparse
const saved = localStorage.getItem('data'); // null が返される可能性
const data = JSON.parse(saved); // null をparseしてエラー

// 関数や undefined を stringify
const withFunction = {name: "太郎", getValue: function() {return 42;}};
const json = JSON.stringify(withFunction); // '{"name":"太郎"}' 関数は消える
```

✅ 正しい例:

```
// エラーハンドリング付きparse
function safeParseJSON(jsonString) {
    try {
        return JSON.parse(jsonString);
```

```
    } catch (error) {
        console.error('JSON解析エラー:', error);
        return null;
    }
}

// null チェック + エラーハンドリング
function loadFromStorage(key) {
    const saved = localStorage.getItem(key);
    if (saved === null) {
        return null; // データなし
    }
    return safeParseJSON(saved);
}

// stringify前の値チェック
function safeSaveToStorage(key, data) {
    if (data === undefined || data === null) {
        localStorage.removeItem(key);
        return;
    }
    try {
        const jsonString = JSON.stringify(data);
        localStorage.setItem(key, jsonString);
    } catch (error) {
        console.error('保存エラー:', error);
    }
}
```

JSONで扱えないデータ型

- **関数**: 無視される
- **undefined**: 無視される（配列の場合は null になる）
- **Symbol**: 無視される
- **Date オブジェクト**: 文字列に変換される（復元時は文字列のまま）
- **正規表現**: 空オブジェクト `{}` になる

深掘り(コラム) - JSONと関連技術

関連技術マップ（優先度順）

🔥 重要度：高 - Date オブジェクトの扱い

関連性: JSON変換でよく問題になるデータ型の代表例

```
// Date オブジェクトの問題と解決例
const originalData = {
    name: "タスク1",
    createdAt: new Date("2025-06-13T10:30:00Z"),
    completed: false
};

// 問題：Dateが文字列になってしまう
const json = JSON.stringify(originalData);
const restored = JSON.parse(json);
console.log(typeof restored.createdAt); // "string" (Dateではない)

// 解決：カスタム変換関数
function stringifyWithDate(data) {
    return JSON.stringify(data, (key, value) => {
        if (value instanceof Date) {
            return {__type: 'Date', value: value.toISOString()};
        }
        return value;
    });
}

function parseWithDate(jsonString) {
    return JSON.parse(jsonString, (key, value) => {
        if (value && value.__type === 'Date') {
            return new Date(value.value);
        }
        return value;
    });
}
```

🔶 重要度：中 - FormData との連携

関連性: フォームデータとJSONオブジェクトの相互変換

```
// FormData と JSON の相互変換例
function formDataToJSON(formData) {
    const object = {};
    formData.forEach((value, key) => {
        // 同じkeyが複数ある場合は配列にする
        if (object[key]) {
            if (!Array.isArray(object[key])) {
                object[key] = [object[key]];
            }
            object[key].push(value);
        } else {
            object[key] = value;
        }
    });
```

```
    return JSON.stringify(object);
}

function jsonToFormData(jsonString) {
    const data = JSON.parse(jsonString);
    const formData = new FormData();

    for (let key in data) {
        if (Array.isArray(data[key])) {
            data[key].forEach(value => formData.append(key, value));
        } else {
            formData.append(key, data[key]);
        }
    }
    return formData;
}
```

🔹 重要度：補足 - XMLHttpRequest/fetch との連携

関連性: サーバー通信でのJSON送受信

```
// サーバー通信でのJSON活用例
async function sendTaskToServer(taskData) {
    try {
        const response = await fetch('/api/tasks', {
            method: 'POST',
            headers: {
                'Content-Type': 'application/json',
            },
            body: JSON.stringify(taskData) // JSON文字列として送信
        });

        const responseData = await response.json(); // 自動でJSONをparse
        return responseData;
    } catch (error) {
        console.error('通信エラー:', error);
        throw error;
    }
}
```

技術の全体像

JSON操作は現代のWebアプリケーション開発において、データの永続化、サーバー通信、設定管理など様々な場面で中核的な役割を果たします。適切なエラーハンドリングと型変換を理解することで、堅牢なアプリケーションを構築できます。

実践応用 - JSONと関連技術の総合活用

組み合わせパターン集

パターン1: JSON + localStorage + 型安全管理

使用場面: 型の整合性を保ちながらデータを永続化する

```
// 型安全なデータ管理システム
class TypeSafeStorage {
    constructor() {
        this.typeValidators = {
            string: (value) => typeof value === 'string',
            number: (value) => typeof value === 'number' && !isNaN(value),
            boolean: (value) => typeof value === 'boolean',
            array: (value) => Array.isArray(value),
            object: (value) => value !== null && typeof value === 'object' && !Array.isArray(value),
            date: (value) => value instanceof Date
        };
    }

    // スキーマ定義に基づく保存
    save(key, data, schema) {
        // バリデーション
        if (!this.validateSchema(data, schema)) {
            throw new Error(`Data does not match schema for key: ${key}`);
        }

        // Date型などの特殊変換
        const processedData = this.processForSave(data, schema);

        try {
            const jsonString = JSON.stringify(processedData);
            localStorage.setItem(key, jsonString);

            // スキーマ情報も保存
            localStorage.setItem(`${key}_schema`, JSON.stringify(schema));

            return { success: true };
        } catch (error) {
            return { success: false, error: error.message };
        }
    }

    // スキーマに基づく復元
    load(key) {
        try {
            const dataString = localStorage.getItem(key);
            const schemaString = localStorage.getItem(`${key}_schema`);
```

```
        if (!dataString || !schemaString) {
            return null;
        }

        const schema = JSON.parse(schemaString);
        const rawData = JSON.parse(dataString);

        // 型復元処理
        const restoredData = this.processForLoad(rawData, schema);

        return restoredData;
    } catch (error) {
        console.error(`Failed to load ${key}:`, error);
        return null;
    }
}

processForSave(data, schema) {
    const result = {...data};

    for (let field in schema) {
        if (schema[field] === 'date' && result[field] instanceof Date) {
            result[field] = {
                __type: 'Date',
                value: result[field].toISOString()
            };
        }
    }

    return result;
}

processForLoad(data, schema) {
    const result = {...data};

    for (let field in schema) {
        if (schema[field] === 'date' && result[field]?.__type === 'Date') {
            result[field] = new Date(result[field].value);
        }
    }

    return result;
}

validateSchema(data, schema) {
    for (let field in schema) {
        const expectedType = schema[field];
        const value = data[field];

        if (value !== undefined && !this.typeValidators[expectedType]?.(value)) {
            return false;
        }
    }
    return true;
}
```

```
}

// 使用例
const storage = new TypeSafeStorage();

const taskSchema = {
    id: 'number',
    text: 'string',
    completed: 'boolean',
    createdAt: 'date',
    tags: 'array'
};

const taskData = {
    id: 1,
    text: "買い物に行く",
    completed: false,
    createdAt: new Date(),
    tags: ["日用品", "食材"]
};

storage.save('task_1', taskData, taskSchema);
const restored = storage.load('task_1');
console.log(restored.createdAt instanceof Date); // true
```

なぜこの組み合わせ？ データの型安全性を保ちながら永続化することで、実行時エラーを大幅に削減

パターン2: JSON + 圧縮 + バージョン管理

使用場面: 大容量データの効率的な保存とバージョン互換性管理

```
// 高度なデータ永続化システム
class AdvancedJSONStorage {
    constructor() {
        this.version = '1.0';
        this.compressionThreshold = 1024; // 1KB以上で圧縮
    }

    // データの保存（圧縮・バージョン情報付き）
    async save(key, data) {
        const envelope = {
            version: this.version,
            timestamp: Date.now(),
            data: data
        };

        try {
            let jsonString = JSON.stringify(envelope);

            // 大容量データの場合は圧縮
            if (jsonString.length > this.compressionThreshold) {
```

```
            jsonString = await this.compress(jsonString);
            envelope.compressed = true;
        }

        localStorage.setItem(key, jsonString);

        // メタデータ保存
        this.saveMetadata(key, {
            size: jsonString.length,
            compressed: envelope.compressed || false,
            timestamp: envelope.timestamp
        });

        return { success: true, size: jsonString.length };
    } catch (error) {
        return { success: false, error: error.message };
    }
}

// データの読み込み（解凍・バージョン互換処理）
async load(key) {
    try {
        let dataString = localStorage.getItem(key);
        if (!dataString) return null;

        // 圧縮データの場合は解凍
        const metadata = this.getMetadata(key);
        if (metadata?.compressed) {
            dataString = await this.decompress(dataString);
        }

        const envelope = JSON.parse(dataString);

        // バージョン互換性チェック
        if (envelope.version !== this.version) {
            envelope.data = this.migrateData(envelope.data, envelope.version, this.version);
        }

        return envelope.data;
    } catch (error) {
        console.error(`Failed to load ${key}:`, error);
        return null;
    }
}

// シンプルな圧縮（実際はpako.jsなどを使用推奨）
async compress(data) {
    // ここでは疑似圧縮（実装例）
    return btoa(data); // Base64エンコード
}

async decompress(data) {
    // ここでは疑似解凍（実装例）
    return atob(data); // Base64デコード
}
```

```
    // データマイグレーション
    migrateData(data, fromVersion, toVersion) {
        if (fromVersion === '0.9' && toVersion === '1.0') {
            // バージョン0.9から1.0への移行例
            if (data.tasks) {
                data.tasks = data.tasks.map(task => ({
                    ...task,
                    priority: task.priority || 'normal', // 新フィールド追加
                    createdAt: task.createdAt || new Date().toISOString()
                }));
            }
        }
        return data;
    }

    saveMetadata(key, metadata) {
        const allMetadata = JSON.parse(localStorage.getItem('__metadata__') || '{}');
        allMetadata[key] = metadata;
        localStorage.setItem('__metadata__', JSON.stringify(allMetadata));
    }

    getMetadata(key) {
        const allMetadata = JSON.parse(localStorage.getItem('__metadata__') || '{}');
        return allMetadata[key];
    }
}
```

技術選択の判断基準

データサイズ小（〜1KB）：基本的なJSON.stringify/parse **データサイズ中（1KB〜100KB）**：エラーハンドリング + 型管理 **データサイズ大（100KB〜）**：圧縮 + 分割保存 **型の整合性重要**：スキーマバリデーション導入 **長期運用**：バージョン管理 + マイグレーション機能

総合実例：実際のプロジェクト想定

```
// 実際のプロジェクトレベルのJSONデータ管理
class ProjectDataManager {
    constructor() {
        this.storage = new AdvancedJSONStorage();
        this.typeStorage = new TypeSafeStorage();

        // アプリケーションスキーマ定義
        this.schemas = {
            userProfile: {
                id: 'string',
                name: 'string',
                email: 'string',
                preferences: 'object',
```

```
            lastLogin: 'date'
        },
        projectData: {
            id: 'string',
            title: 'string',
            tasks: 'array',
            settings: 'object',
            createdAt: 'date',
            updatedAt: 'date'
        }
    };
}

// 統合的なデータ保存
async saveUserProfile(profile) {
    return this.typeStorage.save('userProfile', profile, this.schemas.userProfile);
}

async saveProjectData(project) {
    // 大容量の可能性があるプロジェクトデータは高度な保存を使用
    return this.storage.save('currentProject', project);
}

// エクスポート機能（全データの統合）
async exportAllData() {
    const exportData = {
        userProfile: this.typeStorage.load('userProfile'),
        project: await this.storage.load('currentProject'),
        settings: this.typeStorage.load('appSettings'),
        exportDate: new Date().toISOString(),
        version: '1.0'
    };

    const blob = new Blob([JSON.stringify(exportData, null, 2)], {
        type: 'application/json'
    });

    return blob;
}

// インポート機能（データの復元）
async importData(file) {
    try {
        const text = await file.text();
        const importData = JSON.parse(text);

        // バリデーションとデータ復元
        if (importData.userProfile) {
            await this.saveUserProfile(importData.userProfile);
        }

        if (importData.project) {
            await this.saveProjectData(importData.project);
        }
```

```
            return { success: true };
        } catch (error) {
            return { success: false, error: error.message };
        }
    }
}
```

このような実装により、JSONを中心とした堅牢で拡張性の高いデータ管理システムを構築できます。型安全性、圧縮、バージョン管理など、実際のプロダクション環境で必要とされる機能を全て網羅した実用的なソリューションとなります。

エラーハンドリング（JSON.parse/JSON.stringifyの失敗時対策）

JSON.parseやJSON.stringifyは、無効な文字列や循環参照などで例外（SyntaxError等）を投げることがあります。安全な実装にはtry-catchによる例外処理が不可欠です。

例:

```
// エラーハンドリング付きparse
function safeParseJSON(jsonString) {
    try {
        return JSON.parse(jsonString);
    } catch (error) {
        console.error('JSON解析エラー:', error);
        return null;
    }
}
```

> **より詳しいエラーハンドリングの仕組みやtry-catchの使い方は「補足資料: try-catch詳解」を参照してください。**

関連する補足資料

- **補足資料1: localStorage API詳解** - JSONと組み合わせたデータ永続化
- **補足資料3: データ構造設計** - JSONで表現しやすいデータ構造の設計パターン

- **補足資料4: ブラウザストレージ比較** - 各ストレージでのJSONデータ活用法
- **補足資料: try-catch詳解** - 例外処理の基礎と応用