補足資料: try-catch詳解（エラーハンドリング）

基本(1分) - try-catchとは

概要: JavaScriptでエラーが発生した時にプログラムの異常終了を防ぎ、適切な処理を続行するための構文です。これにより、ユーザーにとって親和性の高いアプリケーションを作成できます。

なぜtry-catchが重要か:

- **プログラムの安定性向上**: エラーでアプリケーション全体が停止することを防ぎます。
- **ユーザー体験の改善**: エラーメッセージを分かりやすく表示し、ユーザーに適切な対処法を提示できます。
- **デバッグの支援**: エラーの詳細情報を取得し、問題の特定を容易にします。

基本構文:

```
try {
  // エラーが発生するかもしれない処理
} catch (error) {
  // エラー発生時の処理
} finally {
  // エラーの有無に関わらず実行される処理（オプション）
}
```

シンプルな例:

```
try {
  const data = JSON.parse('{"name": "太郎", "age": 25}');
  console.log(`名前: ${data.name}, 年齢: ${data.age}`);
} catch (error) {
  console.error('JSON解析エラー:', error.message);
  // ユーザーに分かりやすいメッセージを表示
  alert('データの形式が正しくありません。再度入力してください。');
}
```

よくあるエラーの種類（簡単な紹介）:

- **SyntaxError**: 文法エラー（例: JSON形式が間違っている）

- **TypeError**: 型エラー（例: undefinedのプロパティにアクセス）
- **ReferenceError**: 参照エラー（例: 存在しない変数を使用）

詳細(3分) - try-catchの仕組みと使い方

動作の仕組み

`try-catch`文は以下のフローで動作します：

1. **tryブロックの実行**: 通常の処理を実行します。
2. **エラーの検出**: tryブロック内でエラーが発生すると、そこで処理が中断されます。
3. **catchブロックの実行**: エラーオブジェクトがcatchの引数に渡され、エラー処理が実行されます。
4. **finallyブロックの実行（オプション）**: エラーの有無に関わらず実行されます。

エラーオブジェクトのプロパティ

catchブロックで受け取るエラーオブジェクトには、以下の重要なプロパティがあります：

```
try {
  JSON.parse('{ "name": "太郎", invalid }'); // 意図的に不正なJSON
} catch (error) {
  console.log('エラー名:', error.name);           // "SyntaxError"
  console.log('エラーメッセージ:', error.message); // "Unexpected token i in JSON at position 18"
  console.log('スタックトレース:', error.stack);    // エラーが発生した場所の詳細
}
```

throw文によるカスタムエラー

`throw`文を使用して、独自のエラーを発生させることができます：

```
function validateAge(age) {
  try {
    if (typeof age !== 'number') {
      throw new TypeError('年齢は数値である必要があります');
    }
    if (age < 0) {
      throw new RangeError('年齢は0以上である必要があります');
    }
```

```
    if (age > 150) {
      throw new RangeError('年齢は150以下である必要があります');
    }
    return `年齢: ${age}歳`;
  } catch (error) {
    console.error('年齢検証エラー:', error.message);
    return null;
  }
}

console.log(validateAge(25));     // "年齢: 25歳"
console.log(validateAge('abc'));  // null (エラーメッセージ出力)
console.log(validateAge(-5));     // null (エラーメッセージ出力)
```

主要なエラータイプの詳細

1. SyntaxError (構文エラー)

```
try {
  JSON.parse('{ "name": "太郎" invalid }'); // 不正なJSON構文
} catch (error) {
  if (error instanceof SyntaxError) {
    console.log('JSON形式が正しくありません:', error.message);
  }
}
```

2. TypeError (型エラー)

```
try {
  const obj = null;
  console.log(obj.property); // nullのプロパティにアクセス
} catch (error) {
  if (error instanceof TypeError) {
    console.log('型に関するエラー:', error.message);
  }
}
```

3. ReferenceError (参照エラー)

```
try {
  console.log(undefinedVariable); // 定義されていない変数
} catch (error) {
  if (error instanceof ReferenceError) {
    console.log('変数が定義されていません:', error.message);
```

```
  }
}
```

finallyブロックの活用

`finally`ブロックは、エラーの有無に関わらず必ず実行されます。リソースの解放や後処理に最適です：

```
let file = null;
try {
  file = openFile('data.txt'); // 仮想的なファイル操作
  const content = file.read();
  processData(content);
} catch (error) {
  console.error('ファイル処理エラー:', error.message);
} finally {
  if (file) {
    file.close(); // ファイルが開かれていれば必ず閉じる
    console.log('ファイルを閉じました');
  }
}
```

よくある間違いと注意点

❌ 間違った例1: try-catchで囲まない

```
// エラーでプログラム全体が停止する危険性
const data = JSON.parse(badJsonString);
console.log(data.name); // 上の行でエラーが発生すると、この行は実行されない
```

❌ 間違った例2: エラーを無視する

```
try {
  JSON.parse(jsonString);
} catch (error) {
  // 何もしない（エラーを隠蔽してしまう）
}
```

❌ 間違った例3: 過度に広範囲をtry-catchで囲む

```
try {
  // 大量のコード（エラーの発生箇所が特定しにくくなる）
```

```
  // ...数百行のコード...
} catch (error) {
  console.log('何かしらのエラー'); // どこでエラーが発生したか分からない
}
```

✅ 正しい例1: 適切なエラーハンドリング

```
try {
  const data = JSON.parse(jsonString);
  console.log('パース成功:', data.name);
} catch (error) {
  console.error('JSON解析エラー:', error.message);
  // ユーザーに分かりやすい対処法を提示
  showUserFriendlyMessage('データの形式が正しくありません。');
}
```

✅ 正しい例2: エラータイプに応じた分岐

```
try {
  const data = JSON.parse(jsonString);
} catch (error) {
  if (error instanceof SyntaxError) {
    console.log('JSON形式エラー:', error.message);
  } else {
    console.log('予期しないエラー:', error.message);
  }
}
```

✅ 正しい例3: 必要な箇所のみtry-catchで囲む

```
function safeJsonParse(jsonString) {
  try {
    return JSON.parse(jsonString);
  } catch (error) {
    console.error('JSON解析失敗:', error.message);
    return null; // デフォルト値を返す
  }
}

// 使用箇所
const data = safeJsonParse(userInput);
if (data) {
  // 正常にパースできた場合の処理
  processData(data);
} else {
  // パースに失敗した場合の処理
```

```
  showErrorMessage();
}
```

深掘り（コラム） - エラーハンドリングと関連技術

関連技術マップ（優先度順）

🔥 重要度：高 - Promise/async-awaitでのエラーハンドリング

関連性: 現代のJavaScriptでは非同期処理が多用されるため、Promise/async-awaitでのエラーハンドリングは必須です。

```
// Promise.catch()
fetch('/api/data')
  .then(response => response.json())
  .then(data => console.log(data))
  .catch(error => {
    console.error('通信エラー:', error.message);
  });

// async/await + try-catch
async function fetchData() {
  try {
    const response = await fetch('/api/data');
    const data = await response.json();
    console.log(data);
  } catch (error) {
    console.error('データ取得エラー:', error.message);
  }
}
```

🔥 重要度：高 - Errorオブジェクトのカスタマイズ

関連性: アプリケーション固有のエラー情報を提供するために、カスタムエラークラスを作成することが重要です。

```
// カスタムエラークラスの作成
class ValidationError extends Error {
  constructor(message, field) {
    super(message);
    this.name = 'ValidationError';
    this.field = field;
```

```
  }
}

class NetworkError extends Error {
  constructor(message, statusCode) {
    super(message);
    this.name = 'NetworkError';
    this.statusCode = statusCode;
  }
}

// 使用例
function validateUserInput(userData) {
  if (!userData.email) {
    throw new ValidationError('メールアドレスが必要です', 'email');
  }
  if (!userData.email.includes('@')) {
    throw new ValidationError('有効なメールアドレスを入力してください', 'email');
  }
}

try {
  validateUserInput({ email: '' });
} catch (error) {
  if (error instanceof ValidationError) {
    console.log(`フィールド "${error.field}" のエラー: ${error.message}`);
  }
}
```

🔶 重要度：中 - グローバルエラーハンドラ

関連性: アプリケーション全体での予期しないエラーを捕捉するための最後の砦として重要です。

```
// 同期エラーのグローバルハンドラ
window.addEventListener('error', (event) => {
  console.error('グローバルエラー:', event.error);
  // エラーレポート送信やユーザー通知などの処理
});

// 非同期エラー（未処理のPromise reject）のグローバルハンドラ
window.addEventListener('unhandledrejection', (event) => {
  console.error('未処理のPromise reject:', event.reason);
  event.preventDefault(); // デフォルトのエラー表示を防ぐ
});
```

🔶 重要度：中 - EventListenerでのエラー処理

関連性: DOM操作やユーザーイベント処理でのエラーハンドリングも重要です。

```
// イベントリスナー内でのエラーハンドリング
document.getElementById('submitButton').addEventListener('click', (event) => {
  try {
    const formData = getFormData();
    validateFormData(formData);
    submitData(formData);
  } catch (error) {
    console.error('フォーム処理エラー:', error.message);
    showErrorToUser(error.message);
    event.preventDefault(); // フォーム送信を防ぐ
  }
});
```

🔹 重要度：補足 - ログシステムとの連携

関連性: 本格的なアプリケーションでは、エラーログの収集と分析が重要になります。

```
class Logger {
  static logError(error, context = {}) {
    const logEntry = {
      timestamp: new Date().toISOString(),
      message: error.message,
      stack: error.stack,
      type: error.name,
      context: context
    };

    // 開発環境ではコンソールに出力
    console.error('エラーログ:', logEntry);

    // 本番環境ではサーバーに送信
    // sendToLogServer(logEntry);
  }
}

// 使用例
try {
  riskyOperation();
} catch (error) {
  Logger.logError(error, {
    userId: getCurrentUserId(),
    action: 'データ保存',
    timestamp: Date.now()
  });
}
```

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try-catch**：個別の処理でのエラーハンドリング
2. **Promise/async-await**：非同期処理でのエラーハンドリング
3. **カスタムエラー**：アプリケーション固有のエラー情報提供
4. **グローバルハンドラ**：予期しないエラーの最終的な捕捉
5. **ログシステム**：エラーの記録と分析

実践応用 - エラーハンドリングの総合活用

パターン1: localStorage操作 + try-catch

使用場面: localStorageへのデータ保存・読み込み時のエラーハンドリング

```
class SafeLocalStorage {
  static setItem(key, value) {
    try {
      const jsonString = JSON.stringify(value);
      localStorage.setItem(key, jsonString);
      return true;
    } catch (error) {
      if (error.name === 'QuotaExceededError') {
        console.error('localStorage容量不足:', error.message);
        this.showStorageFullMessage();
      } else if (error instanceof TypeError) {
        console.error('JSONシリアライズエラー:', error.message);
      } else {
        console.error('localStorage保存エラー:', error.message);
      }
      return false;
    }
  }

  static getItem(key, defaultValue = null) {
    try {
      const item = localStorage.getItem(key);
      if (item === null) return defaultValue;
      return JSON.parse(item);
    } catch (error) {
      console.error('localStorage読み込みエラー:', error.message);
      return defaultValue;
    }
  }
```

```
  static showStorageFullMessage() {
    alert('ストレージ容量が不足しています。不要なデータを削除してください。');
  }
}

// 使用例
const userData = { name: '太郎', settings: { theme: 'dark' } };
if (SafeLocalStorage.setItem('userData', userData)) {
  console.log('データ保存成功');
} else {
  console.log('データ保存失敗');
}

const loadedData = SafeLocalStorage.getItem('userData', { name: 'ゲスト' });
console.log('読み込んだデータ:', loadedData);
```

なぜこの組み合わせ？ localStorageは容量制限やブラウザ設定によってエラーが発生する可能性があり、JSON操作でも構文エラーが起こり得るため、try-catchによる安全な操作が不可欠です。

パターン2: fetch API + JSON + try-catch の組み合わせ

使用場面: Web APIからのデータ取得とJSON解析を安全に行う

```
class ApiClient {
  static async fetchTodoList() {
    try {
      const response = await fetch('/api/todos');

      if (!response.ok) {
        throw new NetworkError(
          `サーバーエラー: ${response.status} ${response.statusText}`,
          response.status
        );
      }

      const data = await response.json();
      return { success: true, data };

    } catch (error) {
      if (error instanceof NetworkError) {
        console.error('ネットワークエラー:', error.message);
        return { success: false, error: 'サーバーとの通信に失敗しました' };
      } else if (error instanceof SyntaxError) {
        console.error('JSON解析エラー:', error.message);
        return { success: false, error: 'サーバーからの応答形式が正しくありません' };
      } else {
        console.error('予期しないエラー:', error.message);
        return { success: false, error: '予期しないエラーが発生しました' };
      }
```

```
    }
  }
}

// 使用例
async function loadTodoList() {
  const result = await ApiClient.fetchTodoList();

  if (result.success) {
    console.log('ToDo一覧:', result.data);
    displayTodos(result.data);
  } else {
    console.error('ToDo読み込み失敗:', result.error);
    showErrorMessage(result.error);
  }
}
```

パターン3: ToDoアプリケーションでの包括的エラーハンドリング戦略

使用場面: 実際のToDoアプリケーション全体でのエラーハンドリング

```
class TodoApp {
  constructor() {
    this.todos = [];
    this.setupGlobalErrorHandlers();
    this.initialize();
  }

  setupGlobalErrorHandlers() {
    // 予期しないエラーのグローバルハンドリング
    window.addEventListener('error', (event) => {
      Logger.logError(event.error, { context: 'global-error' });
      this.showUserMessage('予期しないエラーが発生しました。', 'error');
    });

    window.addEventListener('unhandledrejection', (event) => {
      Logger.logError(new Error(event.reason), { context: 'unhandled-promise' });
      this.showUserMessage('通信エラーが発生しました。', 'error');
    });
  }

  async initialize() {
    try {
      // localStorageからデータを復元
      this.todos = SafeLocalStorage.getItem('todos', []);

      // 必要に応じてサーバーと同期
      await this.syncWithServer();

      this.renderTodos();
      console.log('アプリケーション初期化完了');
```

```
  } catch (error) {
    console.error('初期化エラー:', error.message);
    this.showUserMessage('アプリケーションの初期化に失敗しました。', 'error');
  }
}

addTodo(text) {
  try {
    if (!text || text.trim() === '') {
      throw new ValidationError('ToDo内容は空にできません', 'text');
    }

    const newTodo = {
      id: Date.now(),
      text: text.trim(),
      completed: false,
      createdAt: new Date().toISOString()
    };

    this.todos.push(newTodo);

    if (SafeLocalStorage.setItem('todos', this.todos)) {
      this.renderTodos();
      this.showUserMessage('ToDoを追加しました。', 'success');
    } else {
      // 保存失敗時は配列からも削除
      this.todos.pop();
      throw new Error('ToDoの保存に失敗しました');
    }

  } catch (error) {
    if (error instanceof ValidationError) {
      this.showUserMessage(error.message, 'warning');
    } else {
      console.error('ToDo追加エラー:', error.message);
      this.showUserMessage('ToDoの追加に失敗しました。', 'error');
    }
  }
}

async syncWithServer() {
  try {
    const result = await ApiClient.fetchTodoList();
    if (result.success) {
      // サーバーデータとローカルデータをマージ（詳細な実装は省略）
      console.log('サーバーと同期しました');
    }
  } catch (error) {
    // 同期失敗はアプリケーション使用に致命的ではないため、ログのみ
    console.warn('サーバー同期失敗:', error.message);
  }
}

showUserMessage(message, type = 'info') {
  // ユーザー向けメッセージ表示（詳細な実装は省略）
```

```
    console.log(`[${type.toUpperCase()}] ${message}`);
  }

  renderTodos() {
    try {
      // DOM操作によるToDo一覧表示（詳細な実装は省略）
      console.log('ToDo一覧を表示:', this.todos.length, '件');
    } catch (error) {
      console.error('表示エラー:', error.message);
      this.showUserMessage('表示の更新に失敗しました。', 'error');
    }
  }
}

// アプリケーション起動
try {
  const app = new TodoApp();
} catch (error) {
  console.error('アプリケーション起動失敗:', error.message);
  document.body.innerHTML = '<p>アプリケーションを起動できませんでした。ページを再読み込みしてください。</p>';
}
```

技術選択の判断基準まとめ

状況A（個別の関数・処理）の場合: 基本的な `try-catch` を使用し、想定されるエラータイプに応じた分岐処理を行う。

状況B（非同期処理）の場合: `async/await` + `try-catch` または Promise の `.catch()` を使用し、ネットワークエラーや解析エラーに対応する。

状況C（アプリケーション全体）の場合: グローバルエラーハンドラを設置し、個別のエラーハンドリングと組み合わせて包括的な戦略を構築する。

重要な原則:

1. **エラーを隠蔽せず、適切にログとユーザー通知を行う**
2. **エラータイプに応じて異なる対処法を提供する**
3. **リソースの解放やデータの整合性を `finally` で保証する**
4. **ユーザーにとって分かりやすいエラーメッセージを提供する**

関連する補足資料

- JSON操作詳解: `JSON.parse()` と `JSON.stringify()` でのエラーハンドリングの詳細。
- localStorage API詳解: localStorage操作時の具体的なエラーパターンと対策。

- データ構造設計: エラーに強いデータ構造設計の考え方。
- ブラウザストレージ比較: 各ストレージ技術でのエラーハンドリングの違い。

最終更新日: 2024/06/13