JavaScript - async/await

Обработка ошибок, "try..catch"
Пользовательские ошибки
async/await

Обработка ошибок

Неважно, насколько мы хороши в программировании, иногда наши скрипты содержат ошибки. Они могут возникать из-за наших промахов, неожиданного ввода пользователя, неправильного ответа сервера и по тысяче других причин.

Обычно скрипт в случае ошибки «падает» (сразу же останавливается), с выводом ошибки в консоль.

Но есть синтаксическая конструкция try..catch, которая позволяет «ловить» ошибки и вместо падения делать что-то более осмысленное.

Синтаксис «try…catch»

Конструкция try..catch состоит из двух основных блоков: try, и затем catch:

```
try {

  // код...

} catch (err) {

  // обработка ошибки

}
```

Работает она так:

1. Сначала выполняется код внутри блока try {...}.
2. Если в нём нет ошибок, то блок catch(err) игнорируется: выполнение доходит до конца try и потом далее, полностью пропуская catch.
3. Если же в нём возникает ошибка, то выполнение try прерывается, и поток управления переходит в начало catch(err). Переменная err (можно использовать любое имя) содержит объект ошибки с подробной информацией о произошедшем.

try..catch работает только для ошибок, возникающих во время выполнения кода

Чтобы try..catch работал, код должен быть выполнимым. Другими словами, это должен быть корректный JavaScript-код.

JavaScript-движок сначала читает код, а затем исполняет его. Ошибки, которые возникают во время фазы чтения, называются ошибками парсинга. Их нельзя обработать (изнутри этого кода), потому что движок не понимает код.

Таким образом, try..catch может обрабатывать только ошибки, которые возникают в корректном коде. Такие ошибки называют «ошибками во время выполнения», а иногда «исключениями».

```
1  try {
2    {{{{{{{{{{
3  } catch(e) {
4    alert("Движок не может понять этот код, он некорректен");
5  }
```

try..catch работает синхронно

Исключение, которое произойдет в коде, запланированном «на будущее», например в setTimeout, try..catch не поймает

Это потому, что функция выполняется позже, когда движок уже покинул конструкцию try..catch.

Чтобы поймать исключение внутри запланированной функции, try..catch должен находиться внутри самой этой функции:

```
try {
  setTimeout(function() {
    noSuchVariable; // скрипт упадёт тут
  }, 1000);
} catch (e) {
  alert( "не сработает" );
}
```

```
setTimeout(function() {
  try {
    noSuchVariable; // try..catch обрабатывает ошибку!
  } catch {
    alert( "ошибка поймана!" );
  }
}, 1000);
```

Объект ошибки

Когда возникает ошибка, JavaScript генерирует объект, содержащий её детали. Затем этот объект передается как аргумент в блок catch:

name - Имя ошибки. Например, для неопределённой переменной это "ReferenceError".

message - Текстовое сообщение о деталях ошибки.

В большинстве окружений доступны и другие, нестандартные свойства. Одно из самых широко используемых и поддерживаемых – это:

stack - Текущий стек вызова: строка, содержащая информацию о последовательности вложенных вызовов, которые привели к ошибке. Используется в целях отладки.

```
try {
  // ...
} catch(err) { // <-- объект ошибки
  // ...
}
```

```
try {
  lalala; // ошибка, переменная не определена!
} catch(err) {
  alert(err.name); // ReferenceError
  alert(err.message); // lalala is not defined
  alert(err.stack); // ReferenceError: lalala is not defined at (...стек вызовов)

  // Можем также просто вывести ошибку целиком
  // Ошибка приводится к строке вида "name: message"
  alert(err); // ReferenceError: lalala is not defined
}
```

Блок «catch» без переменной

Эта возможность была добавлена в язык недавно. В старых браузерах может понадобиться полифил.

Если нам не нужны детали ошибки, в catch можно ее пропустить:

```
try {
  // ...
} catch { //  <-- без (err)
  // ...
}
```

Оператор «throw»

Оператор throw генерирует ошибку.

Технически в качестве объекта ошибки можно передать что угодно. Это может быть даже примитив, число или строка, но всё же лучше, чтобы это был объект, желательно со свойствами name и message (для совместимости со встроенными ошибками).

В JavaScript есть множество встроенных конструкторов для стандартных ошибок: Error, SyntaxError, ReferenceError, TypeError и другие. Можно использовать и их для создания объектов ошибки.

```
throw <объект ошибки>
```

```
let error = new Error(message);
// или
let error = new SyntaxError(message);
let error = new ReferenceError(message);
// ...
```

Проброс исключения

А что, если в блоке try {...} возникнет другая неожиданная ошибка? Например, программная (неопределённая переменная) или какая-то ещё, а не ошибка, связанная с некорректными данными.

Блок catch должен обрабатывать только те ошибки, которые ему известны, и «пробрасывать» все остальные.

Техника «проброс исключения» выглядит так:

1. Блок catch получает все ошибки.
2. В блоке catch(err) {...} мы анализируем объект ошибки err.
3. Если мы не знаем как её обработать, тогда делаем throw err.

```
let json = '{ "age": 30 }'; // данные неполны
try {

  let user = JSON.parse(json);

  if (!user.name) {
    throw new SyntaxError("Данные неполны: нет имени");
  }

  blabla(); // неожиданная ошибка

  alert( user.name );

} catch(e) {

  if (e.name == "SyntaxError") {
    alert( "JSON Error: " + e.message );
  } else {
    throw e; // проброс (*)
  }

}
```

try…catch…finally

Конструкция try..catch может содержать ещё одну секцию: finally.

Если секция есть, то она выполняется в любом случае:

- после try, если не было ошибок,
- после catch, если ошибки были.

Секцию finally часто используют, когда мы начали что-то делать и хотим завершить это вне зависимости от того, будет ошибка или нет.

```
try {
    ... пробуем выполнить код...
} catch(e) {
    ... обрабатываем ошибки ...
} finally {
    ... выполняем всегда ...
}
```

Область видимости try/catch/finally

Переменные внутри try..catch..finally локальны

Если переменную объявить в блоке, например, в try, то она не будет доступна после него.

finally и return

Блок finally срабатывает при любом выходе из try..catch, в том числе и return.

В примере ниже из try происходит return, но finally получает управление до того, как контроль возвращается во внешний код.

```
function func() {

  try {
    return 1;

  } catch (e) {
    /* ... */
  } finally {
    alert( 'finally' );
  }
}

alert( func() ); // сначала срабатывает alert из finally, а затем этот код
```

try..finally

Конструкция try..finally без секции catch также полезна. Мы применяем её, когда не хотим здесь обрабатывать ошибки (пусть выпадут), но хотим быть уверены, что начатые процессы завершились.

```
function func() {
  // начать делать что-то, что требует завершения (например, измерения)
  try {
    // ...
  } finally {
    // завершить это, даже если все упадёт
  }
}
```

Глобальный catch

В браузере мы можем присвоить функцию специальному свойству window.onerror, которая будет вызвана в случае необработанной ошибки.

window.onerror = function(message, url, line, col, error) {

// ...

};

- message -Сообщение об ошибке.
- url - URL скрипта, в котором произошла ошибка.
- line, col - Номера строки и столбца, в которых произошла ошибка.
- error - Объект ошибки.

Роль глобального обработчика window.onerror обычно заключается не в восстановлении выполнения скрипта – это скорее всего невозможно в случае программной ошибки, а в отправке сообщения об ошибке разработчикам.

```
<script>
  window.onerror = function(message, url, line, col, error) {
    alert(`${message}\n В ${line}:${col} на ${url}`);
  };

  function readData() {
    badFunc(); // Ой, что-то пошло не так!
  }

  readData();
</script>
```

Существуют также веб-сервисы, которые предоставляют логирование ошибок для таких случаев, такие как https://errorception.com или https://www.muscula.com.

Они работают так:

- Мы регистрируемся в сервисе и получаем небольшой JS-скрипт (или URL скрипта) от них для вставки на страницы.
- Этот JS-скрипт ставит свою функцию window.onerror.
- Когда возникает ошибка, она выполняется и отправляет сетевой запрос с информацией о ней в сервис.
- Мы можем войти в веб-интерфейс сервиса и увидеть ошибки.

Пользовательские ошибки, расширение Error

Когда что-то разрабатываем, то нам часто необходимы собственные классы ошибок для разных вещей, которые могут пойти не так в наших задачах. Для ошибок при работе с сетью может понадобиться HttpError, для операций с базой данных DbError, для поиска – NotFoundError и т.д.

Наши ошибки должны поддерживать базовые свойства, такие как message, name и, желательно, stack. Но также они могут иметь свои собственные свойства. Например, объекты HttpError могут иметь свойство statusCode со значениями 404, 403 или 500.

JavaScript позволяет вызывать throw с любыми аргументами, то есть технически наши классы ошибок не нуждаются в наследовании от Error. Но если использовать наследование, то появляется возможность идентификации объектов ошибок посредством obj instanceof Error. Так что лучше применять наследование.

По мере роста приложения, наши собственные ошибки образуют иерархию, например, HttpTimeoutError может наследовать от HttpError и так далее.

Основы расширения Error

Для создания пользовательской ошибки вам нужно создать класс, который наследует от встроенного класса Error. Вот базовый пример:

```
class CustomError extends Error {
  constructor(message) {
    super(message);
    this.name = this.constructor.name; // Устанавливаем имя ошибки
  }
}
```

Пример пользовательской ошибки; Добавление дополнительных свойств.

```
class ValidationError extends Error {
  constructor(message) {
    super(message);
    this.name = 'ValidationError';
  }
}

try {
  throw new ValidationError('Invalid input data');
} catch (error) {
  console.error(`${error.name}: ${error.message}`);
  // ValidationError: Invalid input data
}
```

```
class ValidationError extends Error {
  constructor(message, invalidFields) {
    super(message);
    this.name = 'ValidationError';
    this.invalidFields = invalidFields;
  }
}

try {
  throw new ValidationError('Invalid input data', ['username', 'password']);
} catch (error) {
  console.error(`${error.name}: ${error.message}`);
  // ValidationError: Invalid input data
  console.error(`Invalid fields: ${error.invalidFields.join(', ')}`);
  // Invalid fields: username, password
}
```

Наследование пользовательских ошибок

Вы также можете создавать иерархию пользовательских ошибок, наследуя один пользовательский класс ошибки от другого:

```
class CustomError extends Error {
  constructor(message) {
    super(message);
    this.name = this.constructor.name;
  }
}

class ValidationError extends CustomError {
  constructor(message, invalidFields) {
    super(message);
    this.invalidFields = invalidFields;
  }
}

class AuthenticationError extends CustomError {};

try {
  throw new AuthenticationError('User not authenticated');
} catch (error) {
  console.error(`${error.name}: ${error.message}`);
  // AuthenticationError: User not authenticated
}
```

Поддержка стек-трейса

При создании пользовательских ошибок важно сохранить стек-трейс для отладки. В современных версиях JavaScript это делается автоматически при вызове super(message), но для совместимости со старыми версиями JavaScript можно добавить это вручную.

```
class CustomError extends Error {
  constructor(message) {
    super(message);
    if (Error.captureStackTrace) {
      Error.captureStackTrace(this, this.constructor);
    }
    this.name = this.constructor.name;
  }
}
```

Стек-трейс

Стек-трейс (stack trace) — это список вызовов функций, которые привели к текущему состоянию выполнения программы, обычно включающий информацию о файлах и номерах строк, где происходили вызовы. Когда происходит ошибка, стек-трейс помогает разработчикам понять, как и где произошла ошибка, показывая последовательность вызовов, приведших к ней.

Что включает стек-трейс

- Сообщение об ошибке: Первой строкой выводится сообщение об ошибке, которое было передано при создании объекта ошибки.
- Список вызовов функций: Далее идет список вызовов функций, начиная с того места, где ошибка была выброшена, и заканчивая местом, где выполнение программы началось (или до ближайшего блока try-catch).

Как это помогает

- Диагностика: Стек-трейс показывает точное место, где произошла ошибка, и все функции, которые привели к этой точке. Это помогает быстро найти и исправить проблему.
- Отладка: Разработчики могут видеть последовательность вызовов и могут лучше понять, как их код выполнялся до ошибки.
- Журналирование: Записывая стек-трейс в логи, можно легче отслеживать и анализировать проблемы, возникающие на продакшн-системах.

Пример

Рассмотрим пример кода, который вызывает ошибку, и посмотрим, как выглядит стек-трейс:

Когда выполняется этот код, будет выброшена ошибка в functionC, и стек-трейс будет выглядеть примерно так:

```
Error: Something went wrong
    at functionC (path/to/file.js:9:9)
    at functionB (path/to/file.js:5:3)
    at functionA (path/to/file.js:2:3)
    at Object.<anonymous> (path/to/file.js:13:1)
```

```
function functionA() {
  functionB();
}

function functionB() {
  functionC();
}

function functionC() {
  throw new Error('Something went wrong');
}

try {
  functionA();
} catch (error) {
  console.error(error.stack);
}
```

Обёртывание исключений

Обёртывание исключений (exception wrapping) — это техника в программировании, при которой одна ошибка (исключение) перехватывается и оборачивается в другую, более общую или более специфичную ошибку, прежде чем быть выброшенной дальше. Этот подход помогает абстрагировать и унифицировать обработку ошибок, делая код более удобным для поддержки и читаемым.

Зачем нужно обёртывание исключений?

- Упрощение обработки ошибок: Вместо того, чтобы везде в коде проверять разные виды ошибок, можно проверять только один тип — обернутую ошибку.
- Скрытие деталей реализации: Внешнему коду часто не нужно знать о конкретных деталях внутренних ошибок.
- Улучшение отладки: Обёртывание исключений позволяет сохранять информацию об исходной ошибке, что упрощает ее диагностику и исправление.

Проблема

У нас есть функция readUser, которая выполняет две задачи:

1. Парсит JSON-строку в объект.
2. Проверяет, есть ли в этом объекте нужные поля (валидация).

При этом могут возникнуть два вида ошибок:

- Синтаксическая ошибка при парсинге JSON.
- Ошибка валидации, если в объекте не хватает нужных полей.

Вместо того, чтобы каждый раз в вызывающем коде проверять оба типа ошибок по отдельности, мы хотим упростить обработку ошибок.

Решение: Обёртывание исключений

Мы создаём новый класс ReadError, который будет представлять все ошибки, возникающие в readUser. Когда внутри readUser возникает ошибка, мы "оборачиваем" её в ReadError.

Что происходит в коде

Обработка JSON:

Если при парсинге JSON возникает ошибка (например, синтаксическая ошибка), мы перехватываем её и создаём ReadError, передавая внутрь исходную ошибку (cause).

Валидация:

Если объект не проходит валидацию, мы также создаём ReadError с исходной ошибкой.

Обработка в вызывающем коде:

В вызывающем коде мы обрабатываем только ReadError, не заботясь о деталях (синтаксическая ошибка или ошибка валидации). Мы можем при необходимости получить исходную ошибку из свойства cause.

Этот подход упрощает обработку ошибок в вызывающем коде, так как мы проверяем только один тип ошибки (ReadError), а все детали обрабатываются внутри функции readUser. Это позволяет не писать многократные проверки в каждом месте, где используется readUser.

```
class ReadError extends Error {
  constructor(message, cause) {
    super(message);
    this.cause = cause; // Сохраняем исходную ошибку
    this.name = 'ReadError';
  }
}

class ValidationError extends Error { /*...*/ }
class PropertyRequiredError extends ValidationError { /* ... */ }

function validateUser(user) {
  if (!user.age) {
    throw new PropertyRequiredError("age");
  }

  if (!user.name) {
    throw new PropertyRequiredError("name");
  }
}

function readUser(json) {
  let user;

  try {
    user = JSON.parse(json); // Парсинг JSON
  } catch (err) {
    if (err instanceof SyntaxError) {
      throw new ReadError("Синтаксическая ошибка", err);
    } else {
      throw err;
    }
  }

  try {
    validateUser(user); // Валидация объекта
  } catch (err) {
    if (err instanceof ValidationError) {
      throw new ReadError("Ошибка валидации", err);
    } else {
      throw err;
    }
  }
}

try {
  readUser('{bad json}'); // Ошибка парсинга JSON
} catch (e) {
  if (e instanceof ReadError) {
    alert(e); // Выводим ReadError
    alert("Исходная ошибка: " + e.cause); // Выводим исходную ошибку
  } else {
    throw e;
  }
}
```

Async/await

async и await — это ключевые слова в JavaScript, которые используются для работы с асинхронным кодом.

Использование async и await значительно упрощает работу с асинхронным кодом, делая его более линейным и удобным для чтения и отладки. Эти ключевые слова позволяют избежать вложенных колбэков и упрощают обработку ошибок.

async Функции

Ключевое слово async перед функцией делает её асинхронной. Это означает, что эта функция автоматически возвращает промис, и можно использовать ключевое слово await внутри неё. Вот простой пример:

В этом примере функция myAsyncFunction возвращает строку "Hello, Async!", которая автоматически оборачивается в промис.

```
async function myAsyncFunction() {
  return "Hello, Async!";
}

myAsyncFunction().then((result) => console.log(result)); // Выведет: Hello, Async!
```

await Операторы

Ключевое слово await можно использовать только внутри async функции. Оно приостанавливает выполнение функции до тех пор, пока промис, переданный в await, не разрешится. После этого функция продолжает выполнение, возвращая результат промиса.

```
async function fetchData() {
  let response = await fetch('https://api.example.com/data');
  let data = await response.json();
  return data;
}

fetchData().then((data) => console.log(data));
```

В этом примере:

- await fetch('https://api.example.com/data') приостанавливает выполнение функции до получения ответа от сервера.
- await response.json() приостанавливает выполнение до тех пор, пока ответ не будет преобразован в JSON.

Обработка ошибок

Для обработки ошибок внутри async функции можно использовать стандартные конструкции try и catch.

В этом примере, если запрос не будет успешным, управление перейдет в блок catch, где ошибка будет обработана.

```
async function fetchData() {
  try {
    let response = await fetch('https://api.example.com/data');
    if (!response.ok) {
      throw new Error(`Ошибка: ${response.status}`);
    }
    let data = await response.json();
    return data;
  } catch (error) {
    console.error(error);
  }
}

fetchData();
```

Последовательное выполнение асинхронных операций

Использование await позволяет писать асинхронный код, который выглядит и работает как синхронный, что делает его более читабельным и понятным.

```
async function sequentialTasks() {
  let result1 = await asyncTask1();
  let result2 = await asyncTask2(result1);
  let result3 = await asyncTask3(result2);
  return result3;
}

sequentialTasks().then((result) => console.log(result));
```

Параллельное выполнение асинхронных операций

Если нужно выполнить несколько асинхронных операций параллельно, можно использовать Promise.all. (или другие)

```
async function parallelTasks() {
  let [result1, result2, result3] = await Promise.all([asyncTask1(), asyncTask2(), asyncTask3()]);
  return { result1, result2, result3 };
}

parallelTasks().then((results) => console.log(results));
```

Реализация async

Функция async возвращает промис. Внутри этой функции мы можем использовать await. Ключевое слово await приостанавливает выполнение функции до тех пор, пока промис не разрешится.

Реализуется с помощью Promise-ов и генератора

```
function asyncFunction(generatorFunction) {
  return function(...args) {
    const generator = generatorFunction(...args);

    function handle(result) {
      if (result.done) return Promise.resolve(result.value);

      return Promise.resolve(result.value)
        .then(res => handle(generator.next(res)))
        .catch(err => handle(generator.throw(err)));
    }

    try {
      return handle(generator.next());
    } catch (ex) {
      return Promise.reject(ex);
    }
  };
}
```

Реализация await

Ключевое слово await используется для ожидания разрешения промиса. В данном контексте await — это просто синтаксический сахар для работы с промисами. В нашей реализации async, await будет просто промис, который передаётся в функцию.

- Мы создаём функцию delay, которая возвращает промис, разрешающийся через указанное количество миллисекунд.
- Мы создаём генераторную функцию с использованием asyncFunction, которая возвращает промис и позволяет использовать yield как аналог await.
- Мы вызываем нашу асинхронную функцию и обрабатываем её результат.

```
function delay(ms) {
  return new Promise(resolve => setTimeout(resolve, ms));
}

const asyncExample = asyncFunction(function* () {
  console.log('Начало');
  yield delay(1000);
  console.log('Прошло 1 секунда');
  yield delay(2000);
  console.log('Прошло ещё 2 секунды');
  return 'Завершено';
});

asyncExample().then(result => console.log(result));
```

Объяснение

asyncFunction принимает генераторную функцию и возвращает новую функцию, которая создаёт и управляет генератором.

Внутри новой функции мы создаем генератор и определяем функцию handle, которая обрабатывает результаты выполнения генератора.

handle проверяет, завершён ли генератор (свойство done). Если да, то возвращает разрешенный промис с результатом.

Если генератор не завершен, мы ожидаем разрешения промиса (result.value) и рекурсивно вызываем handle с результатом следующего шага генератора.

Если в генераторе возникает ошибка, она перехватывается и возвращается отклоненный промис.

Таким образом, наша реализация позволяет использовать генераторы для имитации асинхронного поведения, аналогичного async/await.

Ресурсы

Обработка ошибок, "try..catch" - ТЫК

Пользовательские ошибки, расширение Error - ТЫК

Async/await - ТЫК

Своя реализация (на js) async/await - ТЫК