

Раздел

[Обработка ошибок](#)

Навигация по уроку

Расширение Error

Дальнейшее наследование


Обёртывание исключений

Итого

Задачи (1)

Комментарии

Поделиться

[Редактировать на GitHub](#)[🏠 → Язык программирования JavaScript](#)
[→ Обработка ошибок](#) 2-го сентября 2019

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.

Расширение Error

В качестве примера рассмотрим функцию `readUser(json)`, которая должна читать данные пользователя в формате JSON.

Пример того, как может выглядеть корректный `json`:

```
1 let json = `{ "name": "John", "age": 30 }`;
```

Внутри будем использовать `JSON.parse`. При получении некорректного `json` он будет генерировать ошибку `SyntaxError`. Но даже если `json` синтаксически верен, то это не значит, что это будет корректный пользователь, верно? Могут быть пропущены необходимые данные. Например, могут отсутствовать свойства `name` и `age`, которые являются необходимыми для наших пользователей.

Наша функция `readUser(json)` будет не только читать JSON-данные, но и проверять их («валидировать»). Если необходимые поля отсутствуют или данные в неверном формате, то это будет ошибкой. Но не синтаксической ошибкой `SyntaxError`, потому что данные синтаксически корректны. Это будет другая ошибка.

Назовём её ошибкой валидации `ValidationError` и создадим для неё класс. Ошибка этого вида должна содержать информацию о поле, которое является источником ошибки.

Наш класс `ValidationError` должен наследовать от встроенного класса `Error`.

Класс `Error` встроенный, вот его примерный код, просто чтобы мы понимали, что расширяем:

```
1 // "Псевдокод" встроенного класса Error, определённого
2 class Error {
3   constructor(message) {
4     this.message = message;
5     this.name = "Error"; // (разные имена для разных вс
6     this.stack = <стек вызовов>; // нестандартное свойс
7   }
8 }
```

Теперь давайте унаследуем от него `ValidationError` и попробуем новый класс в действии:

Раздел

[Обработка ошибок](#)

Навигация по уроку

Расширение `Error`

Дальнейшее наследование

Обёртывание исключений

Итого

Задачи (1)

Комментарии

Поделиться



Редактировать на GitHub



```
1 class ValidationError extends Error {
2   constructor(message) {
3     super(message); // (1)
4     this.name = "ValidationError"; // (2)
5   }
6 }
7
8 function test() {
9   throw new ValidationError("Упс!");
10 }
11
12 try {
13   test();
14 } catch(err) {
15   alert(err.message); // Упс!
16   alert(err.name); // ValidationError
17   alert(err.stack); // список вложенных вызовов с номер
18 }
```

Обратите внимание: в строке (1) вызываем родительский конструктор. JavaScript требует от нас вызова `super` в дочернем конструкторе, так что это обязательно. Родительский конструктор устанавливает свойство `message`.

Родительский конструктор также устанавливает свойство `name` для `"Error"`, поэтому в строке (2) мы сбрасываем его на правильное значение.

Попробуем использовать его в `readUser(json)`:



```
1 class ValidationError extends Error {
2   constructor(message) {
3     super(message);
4     this.name = "ValidationError";
5   }
6 }
7
8 // Использование
9 function readUser(json) {
10   let user = JSON.parse(json);
11
12   if (!user.age) {
13     throw new ValidationError("Нет поля: age");
14   }
15   if (!user.name) {
16     throw new ValidationError("Нет поля: name");
17   }
18
19   return user;
20 }
21
22 // Рабочий пример с try..catch
23
24 try {
25   let user = readUser('{ "age": 25 }');
26 } catch (err) {
27   if (err instanceof ValidationError) {
28     alert("Некорректные данные: " + err.message); // Не
29   } else if (err instanceof SyntaxError) { // (*)
30     alert("JSON Ошибка Синтаксиса: " + err.message);
31   } else {
32     throw err; // неизвестная ошибка, пробросить исклю
33   }
34 }
```



Раздел

[Обработка ошибок](#)

Навигация по уроку

Расширение Error

Дальнейшее наследование

Обёртывание исключений

Итого

Задачи (1)

Комментарии

Поделиться



[Редактировать на GitHub](#)



Блок `try..catch` в коде выше обрабатывает и нашу `ValidationError`, и встроенную `SyntaxError` из `JSON.parse`.

Обратите внимание, как мы используем `instanceof` для проверки конкретного типа ошибки в строке `(*)`.

Мы можем также проверить тип, используя `err.name`:

```
1 // ...
2 // вместо (err instanceof SyntaxError)
3 } else if (err.name == "SyntaxError") { // (*)
4 // ...
```

Версия с `instanceof` гораздо лучше, потому что в будущем мы собираемся расширить `ValidationError`, сделав его подтипы, такие как `PropertyRequiredError`. И проверка `instanceof` продолжит работать для новых наследованных классов. Так что это на будущее.

Также важно, что если `catch` встречает неизвестную ошибку, то он пробрасывает её в строке `(**)`. Блок `catch` знает, только как обрабатывать ошибки валидации и синтаксические ошибки, а другие виды ошибок (из-за опечаток в коде и другие непонятные) он должен выпустить наружу.

Дальнейшее наследование

Класс `ValidationError` является слишком общим. Много что может пойти не так. Свойство может отсутствовать или иметь неверный формат (например, строка как значение возраста `age`). Поэтому для отсутствующих свойств сделаем более конкретный класс `PropertyRequiredError`. Он будет нести дополнительную информацию о свойстве, которое отсутствует.

```
1 class ValidationError extends Error {
2   constructor(message) {
3     super(message);
4     this.name = "ValidationError";
5   }
6 }
7
8 class PropertyRequiredError extends ValidationError {
9   constructor(property) {
10    super("Нет свойства: " + property);
11    this.name = "PropertyRequiredError";
12    this.property = property;
13  }
14 }
15
16 // Применение
17 function readUser(json) {
18   let user = JSON.parse(json);
19
20   if (!user.age) {
21     throw new PropertyRequiredError("age");
22   }
23   if (!user.name) {
24     throw new PropertyRequiredError("name");
25   }
26
27   return user;
28 }
29
30 // Рабочий пример с try..catch
31
32 try {
33   let user = readUser('{ "age": 25 }');
34 } catch (err) {
35   if (err instanceof ValidationError) {
36     alert("Неверные данные: " + err.message); // Неверны
37     alert(err.name); // PropertyRequiredError
38     alert(err.property); // name
```

Раздел

Обработка ошибок

Навигация по уроку

Расширение Error

Дальнейшее наследование

Обёртывание исключений

Итого

Задачи (1)

Комментарии

Поделиться



Редактировать на GitHub



```
39 } else if (err instanceof SyntaxError) {
40     alert("Ошибка синтаксиса JSON: " + err.message);
41 } else {
42     throw err; // неизвестная ошибка, повторно выбросит
43 }
44 }
```

Новый класс `PropertyRequiredError` очень просто использовать: необходимо указать только имя свойства `new PropertyRequiredError(property)`. Сообщение для пользователя `message` генерируется конструктором.

Обратите внимание, что свойство `this.name` в конструкторе `PropertyRequiredError` снова присвоено вручную. Правда, немного утомительно – присваивать `this.name = <class name>` в каждом классе пользовательской ошибки. Можно этого избежать, если сделать наш собственный «базовый» класс ошибки, который будет ставить `this.name = this.constructor.name`. И затем наследовать все ошибки уже от него.

Давайте назовём его `MyError`.

Вот упрощённый код с `MyError` и другими пользовательскими классами ошибок:

```
1 class MyError extends Error {
2     constructor(message) {
3         super(message);
4         this.name = this.constructor.name;
5     }
6 }
7
8 class ValidationError extends MyError { }
9
10 class PropertyRequiredError extends ValidationError {
11     constructor(property) {
12         super("Нет свойства: " + property);
13         this.property = property;
14     }
15 }
16
17 // name корректное
18 alert( new PropertyRequiredError("field").name ); // Pr
```

Теперь пользовательские ошибки стали намного короче, особенно `ValidationError`, так как мы избавились от строки `"this.name = ..."` в конструкторе.

Обёртывание исключений

Назначение функции `readUser` в приведённом выше коде – это «чтение данных пользователя». В процессе могут возникнуть различные виды ошибок. Сейчас у нас есть `SyntaxError` и `ValidationError`, но в будущем функция `readUser` может расширяться и, возможно, генерировать другие виды ошибок.

Код, который вызывает `readUser`, должен обрабатывать эти ошибки.

Сейчас в нём используются проверки `if` в блоке `catch`, которые проверяют класс и обрабатывают известные ошибки и пробрасывают дальше неизвестные. Но если функция `readUser` генерирует несколько видов ошибок, то мы должны спросить себя: действительно ли мы хотим проверять все типы ошибок поодиночке во всех местах в коде, где вызывается `readUser`?

Часто ответ «Нет»: внешний код хочет быть на один уровень выше всего этого. Он хочет иметь какую-то обобщённую ошибку чтения данных. Почему именно это произошло – часто не имеет значения (об этом говорится в сообщении об ошибке). Или даже лучше, если есть способ получить подробности об ошибке, но только если нам это нужно.

Итак, давайте создадим новый класс `ReadError` для представления таких ошибок. Если ошибка возникает внутри `readUser`, мы её перехватим и

Раздел

[Обработка ошибок](#)

Навигация по уроку

Расширение Error

Дальнейшее наследование

Обёртывание исключений

Итого

Задачи (1)

Комментарии

Поделиться



[Редактировать на GitHub](#)



сгенерируем `ReadError`. Мы также сохраним ссылку на исходную ошибку в свойстве `cause`. Тогда внешний код должен будет только проверить наличие `ReadError`.

Этот код определяет ошибку `ReadError` и демонстрирует её использование в `readUser` и `try..catch`:

```
1 class ReadError extends Error {
2   constructor(message, cause) {
3     super(message);
4     this.cause = cause;
5     this.name = 'ReadError';
6   }
7 }
8
9 class ValidationError extends Error { /*...*/ }
10 class PropertyRequiredError extends ValidationError { /*...*/ }
11
12 function validateUser(user) {
13   if (!user.age) {
14     throw new PropertyRequiredError("age");
15   }
16
17   if (!user.name) {
18     throw new PropertyRequiredError("name");
19   }
20 }
21
22 function readUser(json) {
23   let user;
24
25   try {
26     user = JSON.parse(json);
27   } catch (err) {
28     if (err instanceof SyntaxError) {
29       throw new ReadError("Синтаксическая ошибка", err);
30     } else {
31       throw err;
32     }
33   }
34
35   try {
36     validateUser(user);
37   } catch (err) {
38     if (err instanceof ValidationError) {
39       throw new ReadError("Ошибка валидации", err);
40     } else {
41       throw err;
42     }
43   }
44 }
45
46
47 try {
48   readUser('{bad json}');
49 } catch (e) {
50   if (e instanceof ReadError) {
51     alert(e);
52     // Исходная ошибка: SyntaxError:Unexpected token b
53     alert("Исходная ошибка: " + e.cause);
54   } else {
55     throw e;
56   }
57 }
```

В приведённом выше коде `readUser` работает так, как описано – функция распознаёт синтаксические ошибки и ошибки валидации и выдаёт вместо них ошибки `ReadError` (неизвестные ошибки, как обычно, пробрасываются).

Раздел

[Обработка ошибок](#)

Навигация по уроку

Расширение Error

Дальнейшее наследование

Обёртывание исключений

Итого

Задачи (1)

Комментарии

Поделиться



[Редактировать на GitHub](#)



Внешний код проверяет только `instanceof ReadError`. Не нужно перечислять все возможные типы ошибок

Этот подход называется «обёртывание исключений», потому что мы берём «исключения низкого уровня» и «оборачиваем» их в `ReadError`, который является более абстрактным и более удобным для использования в вызывающем коде. Такой подход широко используется в объектно-ориентированном программировании.

Итого

- Мы можем наследовать свои классы ошибок от `Error` и других встроенных классов ошибок, но нужно позаботиться о свойстве `name` и не забыть вызвать `super`.
- Мы можем использовать `instanceof` для проверки типа ошибок. Это также работает с наследованием. Но иногда у нас объект ошибки, возникшей в сторонней библиотеке, и нет простого способа получить класс. Тогда для проверки типа ошибки можно использовать свойство `name`.
- Обёртывание исключений является распространённой техникой: функция ловит низкоуровневые исключения и создаёт одно «высокоуровневое» исключение вместо разных низкоуровневых. Иногда низкоуровневые исключения становятся свойствами этого объекта, как `err.cause` в примерах выше, но это не обязательно.

✓ Задачи

Наследование от `SyntaxError`

важность: 5

Создайте класс `FormatError`, который наследует от встроенного класса `SyntaxError`.

Класс должен поддерживать свойства `message`, `name` и `stack`.

Пример использования:

```
1 let err = new FormatError("ошибка форматирования");
2
3 alert( err.message ); // ошибка форматирования
4 alert( err.name ); // FormatError
5 alert( err.stack ); // stack
6
7 alert( err instanceof FormatError ); // true
8 alert( err instanceof SyntaxError ); // true (потому что
```

решение

Проводим [курсы по JavaScript и фреймворкам](#).



💬 Комментарии

перед тем как писать...