РАЗРАБОТКА ВЕБ-САЙТОВ*, NODE.JS*, JAVASCRIPT*, БЛОГ КОМПАНИИ MAIL.RU GROUP

У нас проблемы с промисами

ПЕРЕВОД

drfisher 26 октября 2015 в 08:50 112k

Оригинал: Nolan Lawson

Разрешите представить вам перевод статьи Нолана Лоусона «У нас проблемы с промисами», одной из лучших по теме из тех, что мне доводилось читать.

У нас проблемы с промисами

Дорогие JavaScript разработчики, настал момент признать это — у нас проблемы с промисами.

Нет, не с самими промисами. Их реализация по спецификации A+ превосходна. Основная проблема, которая сама предстала передо мной за годы наблюдений за тем, как многие программисты борются с богатыми на промисы API, заключается в следующем:

— Многие из нас используют промисы без действительного их понимания.

Если вы мне не верите, решите такую задачку:

Вопрос: В чем разница между этими четырьмя вариантами

использования промисов?

```
doSomething().then(function () {
  return doSomethingElse();
});

doSomething().then(function () {
  doSomethingElse();
});

doSomething().then(doSomethingElse());

doSomething().then(doSomethingElse);
```

Если вы знаете ответ, то разрешите вас поздравить — по части промисов вы ниндзя. Пожалуй, дальше этот пост вам можно не читать.

Остальным 99,99% из вас я спешу сказать, чтобы вы не расстраивались, вы в хорошей компании. Никто из тех, кто ответил на мой твит, не смог решить задачу. Даже я был очень удивлен ответом на 3-й вопрос. Да-да, несмотря на то, что это я его задал!

Ответ на задачу приведен в самом конце статьи, но сначала я хотел бы объяснить, почему промисы в первом приближении такие коварные, и почему многие из нас, новички и эксперты, попадаются в их ловушки. Также я хочу предложить вам свое решение, один необычный трюк, который поможет лучше понять промисы. И, разумеется, я верю, что после моего объяснения они уже не покажутся вам такими сложными.

Прежде, чем начать, давайте обозначим некоторые моменты.

Почему промисы?

Если вы читаете статьи о промисах, то часто будете встречать отсылки на «пирамиду зла» («pyramide of doom» в ориг.), образованную ужасным кодом на колбэках, который растягивает страницу направо за экран.

Промисы действительно решают эту проблему, но это нечто большее, чем просто уменьшение отступов. Как объясняется в замечательной беседе «Спасение из ада колбэков», настоящая их проблема в том, что они лишают нас возможности использовать инструкции **return** и **throw**. Вместо этого логика наших программ основана на использовании побочных эффектов, когда одна функция вызывает другую.

Также колбэки делают что-то действительно зловещее, они лишают нас стека, того, что в языках программирования принимается как должное. Писать код без стека — это все равно, что вести автомобиль без педали тормоза. Вы не представляете, насколько она важна до тех пор, пока она вам действительно не понадобится и ее не окажется на месте.

Весь смысл промисов в том, чтобы вернуть нам основы языка, потерянные в момент нашего перехода на асинхронность: **return**, **throw** и стек. Но вы должны знать, как правильно использовать промисы, чтобы подняться в этом на более высокий уровень.

Ошибки новичков

Кто-то пытается объяснить промисы в виде мультика или, говоря словами: *«О! Это штука, которую вы можете создать и передавать всюду, а она собой символизирует какое-то значение, получаемое и возвращаемое асинхронно»*.

Я не нахожу такое объяснение достаточно полезным. Для меня промисы — это всегда часть структуры кода, его потока выполнения.

Небольшое отступление: термин «промисы» для разных людей несет в себе разный смысл. В этой статье я буду рассказывать об официальной спецификации, доступной в современных браузерах как **window.Promise**. Для тех браузеров, которые не имеют **window.Promise**, есть хороший полифил с нахальным названием Lie (ложь), содержащий минимальную реализацию спецификации.

Ошибка новичка №1 — «пирамида зла» из промисов

Глядя, как люди используют PouchDB, API которого сильно завязано на промисах, я вижу немало плохих паттернов их использования. Вот наиболее распространенный пример:

```
remotedb.allDocs({
  include_docs: true,
  attachments: true
}).then(function (result) {
  var docs = result.rows;
  docs.forEach(function(element) {
    localdb.put(element.doc).then(function(response) {
      alert("Pulled doc with id " + element.doc._id + " and added to
```

```
local db.");
    }).catch(function (err) {
      if (err.status == 409) {
        localdb.get(element.doc._id).then(function (resp) {
          localdb.remove(resp._id, resp._rev).then(function (resp) {
// и так далее…
```

Да, получается, что мы можем использовать промисы так, будто это колбэки, и да, это все равно, что стрелять из пушки по воробьям.

Если вы думаете, что подобные ошибки совершают только абсолютные новички, я вас удивлю — код примера выше взят из официального блога разработчиков BlackBerry! От старой привычки использовать колбэки избавиться трудно.

Вот вариант получше:

```
remotedb.allDocs(...)
  .then(function (resultOfAllDocs) {
    return localdb.put(...);
  })
  .then(function (resultOfPut) {
    return localdb.get(...);
  })
  .then(function (resultOfGet) {
    return localdb.put(...);
  })
  .catch(function (err) {
    console.log(err);
  });
```

В примере выше использовались составные промисы («composing promises» в ориг.) — одна из сильнейших сторон промисов. Каждая последующая функция будет вызвана, когда предыдущий промис

«зарезолвится», и вызвана она будет с результатом работы предыдущего промиса. Подробности будут ниже.

Ошибка новичка №2 — как мне использовать forEach() с промисами?

Это тот момент, когда понимание промисов большинством людей начинает сдавать. Они хорошо знакомы с итераторами **forEach**, **for** или **while**, но не имеют ни малейшего представления, как сочетать их с промисами. Тогда рождается что-то подобное:

```
// Я хочу применить remove() ко всем документам
db.allDocs({include_docs: true})
  .then(function (result) {
    result.rows.forEach(function (row) {
      // Метод remove возвращает promise
      db.remove(row.doc);
    });
  })
  .then(function () {
    // А здесь я наивно уверен, что все документы уже удалены!
  });
```

Что не так с этим кодом? Проблема в том, что первая функция возвращает **undefined**, а значит вторая не ждет окончания выполнения **db.remove()** для всех документов. На самом деле она вообще ничего не ждет и выполнится, когда будет удалено любое число документов, а может и ни одного.

Это очень коварная ошибка, потому что поначалу вы можете ее даже не заметить, особенно, если документы будут удаляться достаточно быстро для обновления интерфейса. Бага может всплывать только в редких случаях, не во всех браузерах, а значит,

выявить и устранить ее будет практически невозможно.

Подводя итог, скажу, что конструкции типа **forEach**, **for** и **while** *«не те дроны, что вы ищете»*. Вам нужен **Promise.all()**:

```
db.allDocs({include_docs: true})
  .then(function (result) {
    var arrayOfPromises = result.rows.map(function (row) {
      return db.remove(row.doc);
    });
    return Promise.all(arrayOfPromises);
  })
  .then(function (arrayOfResults) {
    // Вот теперь все документы точно удалены!
  });
```

Что здесь происходит? **Promise.all()** принимает в качестве аргумента массив промисов и возвращает новый промис, который «зарезолвится», только когда все документы будут удалены. Это асинхронный эквивалент цикла **for**.

Также промис из **Promise.all()** передаст в следующую функцию массив результатов, что может быть очень удобно, если вы, например, не удаляете документы, а получаете данные сразу из нескольких источников. Если хотя бы один промис из массива, переданного в **Promise.all()** «зареджектится», то и результирующий промис перейдет в состояние **rejected**.

Ошибка новичка №3 — забываем добавлять .catch()

Это еще одна распространенная ошибка — блаженно верить, что ваши промисы никогда не вернут ошибку. Многие разработчики

просто забывают добавлять **catch()** куда-либо в своем коде.

К сожалению, часто это означает, что ошибки будут «проглочены». Вы даже никогда не узнаете, что они были — особая боль при отладке приложения.

Чтобы избежать этого неприятного сценария, я взял за правило, которое затем переросло в привычку, всегда добавлять в конец моей цепочки промисов метод **catch()**:

```
somePromise().then(function () {
    return anotherPromise();
  })
  .then(function () {
    return yetAnotherPromise();
  })
  // простое и полезное окончание цепочки промисов:
  .catch(console.log.bind(console));
```

Даже если вы гарантированно, стопроцентно не ожидаете каких-либо ошибок, добавление **catch()** будет разумным решением. Потом, если вдруг ваше предположение насчет ошибок не оправдается, вы скажете себе спасибо.

Ошибка новичка №4 — использование «deferred»

Такую ошибку я вижу постоянно и не хочу даже повторять название этого объекта, опасаясь, что он, подобно Битлджусу из одноименного фильма, только того и ждет, чтобы увеличить число случаев своего использования.

Вкратце, в своем развитии промисы прошли долгую историю. У JavaScript сообщества ушло много времени на то, чтобы реализовать их правильно. Поначалу jQuery и Angular повсеместно использовали паттерн deferred-объектов, который впоследствии был заменен на спецификацию промисов ES6, в основе которой лежали «хорошие» библиотеки Q, When, RSVP, Bluebird, Lie и другие.

В общем, если вы вдруг написали это слово в своем коде (я не повторю его в третий раз!), знайте — вы что-то делаете не так. Ниже рецепт, как этого избежать.

Большинство «промисных» библиотек дают вам возможность импортировать промисы из других библиотек. Например, модуль **$q** из Angular позволяет вам обернуть не-$q промисы при помощи **$q.when()**. То есть пользователи Angular могут оборачивать промисы из PouchDB так:

```
// это все, что нужно:
$q.when(db.put(doc)).then(/* ... */);
```

Другой путь — использование паттерна раскрытия конструктора («revealing constructor pattern» в ориг.). Он удобен для оборачивания API, не использующего промисы. Например, чтобы обернуть основанное на колбэках API Node.js, вы можете сделать следующее:

```
new Promise(function (resolve, reject) {
  fs.readFile('myfile.txt', function (err, file) {
    if (err) {
```

```
      return reject(err);
    }
    resolve(file);
  });
}).then(/* ... */)
```

Готово! Мы расправились с ужасным defer… Ах, чуть не произнес в третий раз! :)

Ошибка новичка №5 — использование внешних функций вместо возвращения результата

Что не так с этим кодом?

```
somePromise().then(function () {
    someOtherPromise();
  })
  .then(function () {
    // Ох, я надеюсь someOtherPromise «зарезолвился»…
    // Осторожно, спойлер: нет, не «зарезолвился».
  });
```

Хорошо, сейчас идеальный момент для того, чтобы поговорить обо всем том, что вы вообще должны знать о промисах.

Серьезно, это тот самый трюк, поняв который, вы сами сможете избежать всех тех ошибок, о которых мы говорили. Вы готовы?

Как я уже упоминал, магия промисов в том, что они возвращают нам драгоценные **return** и **throw**. Но что это означает на практике?

Каждый промис предоставляет вам метод **then()** (а еще **catch()**, который на практике просто «сахар» для **then(null, …)**). И вот мы

внутри функции **then()**:

```
somePromise().then(function () {
  // Вау, мы внутри функции then()!
});
```

Что мы можем тут сделать? Три вещи:

1. Вернуть (**return**) другой промис
2. Вернуть (**return**) синхронное значение (или **undefined**)
3. Выдать (**throw**) синхронную ошибку

Вот и все, весь трюк. Поймете его — поймете промисы. Давайте теперь разберем подробно каждый из пунктов.

1. Вернуть другой промис

Это частый паттерн, который вы могли видеть во всевозможной литературе о промисах, а также в примере с составными промисами выше:

```
getUserByName('nolan').then(function (user) {
    // Функция getUserAccountById возвращает promise,
    // результат которого попадет в следующий then
    return getUserAccountById(user.id);
  })
  .then(function (userAccount) {
    // Я знаю все о пользователе!
  });
```

Обратите внимание, что я именно возвращаю второй промис, использую **return**. Использование здесь **return** — это ключевой

момент. Если бы я просто вызвал **getUserAccountById**, то да, был бы запрос за данными пользователя, был бы получен результат, который нигде бы не пригодился — следующий **then** получил бы **undefined** вместо желанного **userAccount**.

2. Вернуть синхронное значение (или undefined)

Возвращение **undefined** в качестве результата — частая ошибка. А вот возвращение какого-либо синхронного значения — отличный способ преобразовать синхронный код в цепочку промисов. Допустим, у нас в памяти есть кэш данных о пользователях. Мы можем:

```
getUserByName('nolan').then(function (user) {
    if (inMemoryCache[user.id]) {
      // Данные этого пользователя уже есть,
      // возвращаем сразу
      return inMemoryCache[user.id];
    }
    // А вот про этого пока не знаем,
    // вернем промис запроса
    return getUserAccountById(user.id);
  })
  .then(function (userAccount) {
    // Я знаю все о пользователе!
  });
```

Разве не круто? Второй функции в цепочке не важно, откуда взялись данные, из кэша или как результат запроса, а первая вольна вернуть или синхронное значение сразу, или асинхронный промис, который уже в свою очередь вернет синхронное значение.

К сожалению, если вы не использовали **return**, то функция все равно вернет значение, но им будет уже не результат вызова

вложенной функции, а бесполезный **undefined**, возвращаемый в таких случаях по-умолчанию.

Для себя я ввел правило, которое затем переросло в привычку — всегда использовать **return** внутри then или выдавать ошибку при помощи **throw**. Я рекомендую вам поступать так же.

3. Выдавать синхронную ошибку

Вот мы и подошли к **throw**. Здесь промисы начинают сиять еще ярче. Предположим, мы хотим выдать (**throw**) синхронную ошибку, если пользователь не авторизован. Это просто:

```
getUserByName('nolan').then(function (user) {
    if (user.isLoggedOut()) {
      // Пользователь вышел - выдаем ошибку!
      throw new Error('user logged out!');
    }
    if (inMemoryCache[user.id]) {
      // Данные этого пользователя уже есть,
      // возвращаем сразу
      return inMemoryCache[user.id];
    }
    // А вот про этого пока не знаем,
    // вернем промис запроса
    return getUserAccountById(user.id);
  })
  .then(function (userAccount) {
    // Я знаю все о пользователе!
  })
  .catch(function (err) {
    // Упс, ошибка, но мы к ней готовы!
  });
```

Наш **catch()** получит синхронную ошибку, если пользователь не авторизован, или асинхронную, если любой из промисов выше

перейдет в состояние **rejected**. И снова, функции в **catch** без разницы, была ошибка синхронной или асинхронной.

Это особенно удобно для отлавливания ошибок во время разработки. Например, формирование объекта из строки при помощи **JSON.parse()** где-либо внутри **then()** может выдать ошибку, если **json** невалидный. С колбэками она будет «проглочена», но при помощи метода **catch()** мы без труда сможем ее обработать.

Продвинутые ошибки

Хорошо, теперь, когда вы выучили главный трюк промисов, давайте поговорим о крайних случаях. Потому что крайние случаи есть всегда.

Эту категорию ошибок я называю «продвинутые», потому что встречал их только в коде хорошо знакомых с промисами программистов. Обсудить подобные ошибки мы должны для того, чтобы разобрать задачку, которую я опубликовал в самом начале статьи.

Продвинутая ошибка №1 — не знаем о Promise.resolve()

Я уже показывал выше, насколько удобны промисы при оборачивании синхронной логики в асинхронный код. Вероятно, вы могли замечать за собой что-то похожее:

```
new Promise(function (resolve, reject) {
  resolve(someSynchronousValue);
}).then(/* ... */);
```

Имейте в виду, вы можете написать то же самое гораздо короче:

```
Promise.resolve(someSynchronousValue).then(/* ... */);
```

Также такой подход очень удобен для отлавливания любых синхронных ошибок. Он настолько удобен, что я использую его почти во всех методах API, возвращающих промисы:

```
function somePromiseAPI() {
  return Promise.resolve()
    .then(function () {
      doSomethingThatMayThrow();
      return 'foo';
    })
    .then(/* ... */);
}
```

Просто запомните, любой код, который может выдать синхронную ошибку — потенциальная проблема при отладке из-за «проглоченных» ошибок. Но если вы обернете его в **Promise.resolve()**, то можете быть уверены, что поймаете ее при помощи **catch()**.

Еще есть **Promise.reject()**. Его можно использовать для возвращения промиса в статусе **rejected**:

```
Promise.reject(new Error('Какая-то ужасная ошибка'));
```

Продвинутая ошибка №2 — catch() не одно и то же с then(null, …)

Чуть выше я упоминал, что **catch()** — это просто «сахар». Два примера ниже эквивалентны:

```
somePromise().catch(function (err) {
  // Обрабатываем ошибку
});

somePromise().then(null, function (err) {
  // Обрабатываем ошибку
});
```

Однако, примеры ниже уже не «равны»:

```
somePromise().then(function () {
    return someOtherPromise();
  })
  .catch(function (err) {
    // Обработка ошибка
  });

somePromise().then(function () {
  return someOtherPromise();
}, function (err) {
  // Обработка ошибки
});
```

Если вы задумались, почему примеры выше «не равны», посмотрите внимательно, что произойдет, если в первой функции возникнет ошибка:

```
somePromise().then(function () {
    throw new Error('oh noes');
  })
  .catch(function (err) {
    // Ошибка поймана! :)
  });

somePromise().then(function () {
  throw new Error('oh noes');
}, function (err) {
  // Ошибка? Какая ошибка? O_o
});
```

Получается, что, если вы используете формат **then(resolveHandler, rejectHandler)**, то **rejectHandler** по факту не может поймать ошибку, возникшую внутри функции **resolveHandler**.

Зная эту особенность, для себя я ввел правило никогда не использовать вторую функцию в методе **then()**, а взамен всегда добавлять обработку ошибок ниже в виде **catch()**. Исключение у меня только одно — асинхронные тесты в Mocha, в случаях, когда я намеренно жду ошибку:

```
it('should throw an error', function () {
  return doSomethingThatThrows().then(function () {
    throw new Error('I expected an error!');
  }, function (err) {
    should.exist(err);
  });
});
```

К слову, Mocha и Chai — отличная комбинация для тестирования основанного на промисах API.

Продвинутая ошибка №3 — промисы против фабрик промисов

Допустим, вы хотите выполнить серию промисов один за другим, последовательно. Вы хотите что-то вроде **Promise.all()**, но такой, чтобы **не** выполнял промисы параллельно.

Сгоряча вы можете написать что-то подобное:

```
function executeSequentially(promises) {
  var result = Promise.resolve();
  promises.forEach(function (promise) {
    result = result.then(promise);
  });
  return result;
}
```

К сожалению, пример выше не будет работать так, как задумывалось. Промисы из списка, переданного в **executeSequentially()**, все равно начнут выполняться параллельно.

Причина в том, что по спецификации промис начинает выполнять заложенную в него логику сразу после создания. Он не будет ждать. Таким образом, не сами промисы, а массив фабрик промисов — это то, что действительно нужно передать в **executeSequentially**:

```
function executeSequentially(promiseFactories) {
  var result = Promise.resolve();
  promiseFactories.forEach(function (promiseFactory) {
    result = result.then(promiseFactory);
  });
```

```
  return result;
}
```

Я знаю, вы сейчас думаете: *«Кто, черт возьми, этот Java программист, и почему он рассказывает нам о фабриках?»*. На самом деле фабрика — это простая функция, возвращающая промис:

```
function myPromiseFactory() {
  return somethingThatCreatesAPromise();
}
```

Почему этот пример будет работать? А потому, что наша фабрика не создаст промис до тех пор, пока до него не дойдет очередь. Она работает именно как **resolveHandler** для **then()**.

Посмотрите внимательно на функцию **executeSequentially()** и мысленно замените ссылку на **promiseFactory** ее содержимым — сейчас над вашей головой должна радостно вспыхнуть лампочка :)

Продвинутая ошибка №4 — что, если я хочу результат двух промисов?

Часто бывает, что один промис зависит от другого, а нам на выходе нужны результаты обоих. Например:

```
getUserByName('nolan').then(function (user) {
    return getUserAccountById(user.id);
  })
  .then(function (userAccount) {
    // Стойте, мне еще и объект «user» нужен!
  });
```

Желая оставаться хорошими JavaScript разработчиками, мы возможно захотим вынести на более высокий уровень видимости переменную **user**, дабы не создавать «пирамиду зла».

```
var user;
getUserByName('nolan').then(function (result) {
    user = result;
    return getUserAccountById(user.id);
  })
  .then(function (userAccount) {
    // Хорошо, теперь есть и «user», и «userAccount»
  });
```

Это работает, но лично я считаю, что код «попахивает». Мое решение — отодвинуть в сторону предубеждения и сделать осознанный шаг в сторону «пирамиды»:

```
getUserByName('nolan').then(function (user) {
    return getUserAccountById(user.id)
      .then(function (userAccount) {
        // Хорошо, теперь есть и «user», и «userAccount»
      });
});
```

… по крайней мере, временный шаг. Если же вы почувствуете, что отступы увеличиваются и пирамида начинает угрожающе расти, сделайте то, что JavaScript разработчики делали испокон веков — создайте функцию и используйте ее по имени.

```
function onGetUserAndUserAccount(user, userAccount) {
  return doSomething(user, userAccount);
}
```

```
function onGetUser(user) {
  return getUserAccountById(user.id)
    .then(function (userAccount) {
      return onGetUserAndUserAccount(user, userAccount);
    });
}

getUserByName('nolan')
  .then(onGetUser)
  .then(function () {
    // К этому моменту функция doSomething() выполнилась,
    // а отступы не выросли — пирамидой и не пахнет
  });
```

По мере того, как код будет усложняться, вы обратите внимание, что все большая его часть преобразовывается в именованные функции, а сама логика приложения начинает приобретать вид, приносящий эстетическое удовольствие:

```
putYourRightFootIn()
  .then(putYourRightFootOut)
  .then(putYourRightFootIn)
  .then(shakeItAllAbout);
```

Вот для чего нам промисы.

Продвинутая ошибка №5 — «проваливание» сквозь промисы

Наконец, именно на эту ошибку я намекал, предлагая вам решить задачку в начале статьи. Это очень редкий случай. Возможно, он никогда не появится в вашем коде. Повстречав его, я определенно удивился.

Как вы думаете, что выведет в консоль этот код?

```
Promise.resolve('foo')
  .then(Promise.resolve('bar'))
  .then(function (result) {
    console.log(result);
  });
```

Если вы думаете, что это будет bar, вы ошиблись. В консоли появится foo!

Причина в том, что, когда вы передаете в **then()** что-то отличное от функции (например, промис), это интерпретируется как **then(null)** и в следующий по цепочке промис «проваливается» результат предыдущего. Проверьте сами:

```
Promise.resolve('foo')
  .then(null)
  .then(function (result) {
    console.log(result);
  });
```

Добавьте сколько угодно **then(null)**, результат останется прежним — в консоли вы увидите foo.

Данный пример возвращает нас к выбору между промисами и фабриками промисов. Мы разбирали его выше. Если коротко, вы можете передавать прямо в **then()** промис, но результат будет совсем не тем, что вы ожидали. Метод **then()** ожидает функцию. Чтобы ожидания сбылись, нужно переписать пример как-то так:

```
Promise.resolve('foo')
  .then(function () {
    return Promise.resolve('bar');
  })
  .then(function (result) {
    console.log(result);
  });
```

В консоли вы увидите bar, как и ожидали.

Запоминаем: в метод **then()** передаем только функции.

Решение задачи

Теперь, когда мы о промисах знаем все или, по крайней мере, близки к этому, мы должны решить задачу, которую я опубликовал выше.

Вот ответы на все пазлы с визуализацией для лучшего понимания.

Пазл №1

```
doSomething().then(function () {
    return doSomethingElse();
  })
  .then(finalHandler);
```

Ответ:

```
doSomething
|-----------------|
                  doSomethingElse(undefined)
                  |-----------------|
```

```
finalHandler(resultOfDoSomethingElse)
                                        |------------------|
```

Пазл №2

```
doSomething().then(function () {
    doSomethingElse();
  })
  .then(finalHandler);
```

Ответ:

```
doSomething
|-----------------|
                  doSomethingElse(undefined)
                  |------------------|
                  finalHandler(undefined)
                  |------------------|
```

Пазл №3

```
doSomething().then(doSomethingElse())
  .then(finalHandler);
```

Ответ:

```
doSomething
|-----------------|
doSomethingElse(undefined)
|---------------------------------|
                  finalHandler(resultOfDoSomething)
                  |------------------|
```

Пазл №4

```
doSomething().then(doSomethingElse)
  .then(finalHandler);
```

Ответ:

```
doSomething
|-----------------|
                  doSomethingElse(resultOfDoSomething)
                  |-----------------|

finalHandler(resultOfDoSomethingElse)
                                     |-----------------|
```

Если вы не поняли объяснения, то я советую вам перечитать статью еще раз, или самостоятельно написать функции **doSomething()** и **doSomethingElse()**, и поэкспериментировать с ними в браузере. Подразумевается, что эти функции возвращают промисы с какими-то результирующими данными.

Также обратите внимание на мой список полезных заготовок.

Заключительное слово о промисах

Промисы очень хороши. Если вы до сих пор используете колбэки, я настойчиво рекомендую вам переключиться на промисы. Ваш код станет меньше, элегантнее, а значит, более простым для понимания.

Если вы мне не верите, держите пруф-линк — рефакторинг

PounchDB модуля map/reduce с заменой колбэков на промисы. Результат: 290 добавленных строк, 555 удаленных. По-случайности, автором всех этих прежних жутких колбэков был… я. Так что это стало для меня первым из освоенных преимуществ промисов.

Я уже говорил, промисы великолепны. Это правда, что они лучше колбэков. Тем не менее, это все равно, что выбирать между пинком в живот и ударом в зубы. Да, что-то из этого лучше, но еще лучше избежать обоих вариантов. Промисы все еще трудны для понимания, и можно попасть в ситуацию, когда результат работы кода намного отличается от задуманного. Даже опытные разработчики могут попасть в коварную ловушку. Основная проблема в том, что промисы, хоть и реализуют паттерны, схожие с синхронным кодом, на деле совсем не равны им.

Ждем async/await

В своей статье «Укрощение асинхронного чудовища с ES7» я упоминал **async/await** и то, как они глубже интегрировали промисы в язык. Вместо написания псевдо-синхронного кода (с фейковым методом **catch()**, который как из **try/catch**, но не совсем, ES7 позволит нам использовать настоящие **try/catch/return**.

Это огромное благо для JavaScript, как для языка, потому что анти-паттерны использования промисов по-прежнему будут возникать, пока наши инструменты не будут сообщать нам о сделанных ошибках.

В качестве примера из истории JavaScript, я думаю, что JSLint и JSHint сделали для сообщества гораздо больше, чем «Хорошие стороны JavaScript», хоть в целом суть у них похожа. Разница в том, что в первом случае вам говорят о конкретной ошибке, которую вы сделали в определенной строке, а во втором вы читаете большую книгу о том, какие ошибки делают другие разработчики.

Красота **async/await** в основном в том, что ваши ошибки проявят себя сразу при синтаксическом анализе в интерпретаторе JS, а не когда-то возможно где-то во время исполнения кода. До тех же пор полезным будет представление о возможностях промисов, о том, как их использовать в ES5 и ES6.

Как и книга «Хорошие стороны JavaScript», эта статья имеет малый эффект. Вероятно однажды вы сможете дать на нее ссылку своему коллеге, который найдет в себе силы честно признать:

— У меня проблемы с промисами!

Комментарии (121)

Похожие публикации

Функциональное программирование в JavaScript с практическими примерами

47

ПЕРЕВОД

AloneCoder • 28 апреля 2017 в 14:20

Вы не знаете Node: краткий обзор основных возможностей

32

ПЕРЕВОД

dedokOne • 11 мая 2016 в 11:04

10 основных ошибок при разработке на Node.js

45

ПЕРЕВОД

ZaValera • 16 апреля 2015 в 18:42

Популярное за сутки

Наташа — библиотека для извлечения структурированной информации из текстов на русском языке

14

alexkuku • вчера в 16:12

Unit-тестирование скриншотами: преодолеваем звуковой барьер. Расшифровка доклада 4

lahmatiy • вчера в 13:05

Люди не хотят чего-то действительно нового — они хотят привычное, но сделанное иначе 25

ПЕРЕВОД

Smileek • вчера в 10:32

Руководство по SEO JavaScript-сайтов. Часть 2. Проблемы, эксперименты и рекомендации 2

ПЕРЕВОД

ru_vds • вчера в 12:04

Как адаптировать игру на Unity под iPhone X к апрелю 0

P1CACHU • вчера в 16:13

Лучшее на Geektimes

Стивен Хокинг, автор «Краткой истории времени», умер на 77 году жизни 33

HostingManager • вчера в 13:49

Обзор рынка моноколес 2018 70

lozga • вчера в 06:58

«Битва за Telegram»: 35 пользователей подали в суд на ФСБ

40

alizar • вчера в 15:14

Стивен Хокинг и его работа — что дал ученый человечеству?

8

marks • вчера в 14:46

Sunlike — светодиодный свет нового поколения

17

AlexeyNadezhin • вчера в 20:32