



681,62

Рейтинг

RUVDS.com

RUVDS — хостинг VDS/VPS серверов



ru_vds 10 апреля в 14:03 Разработка

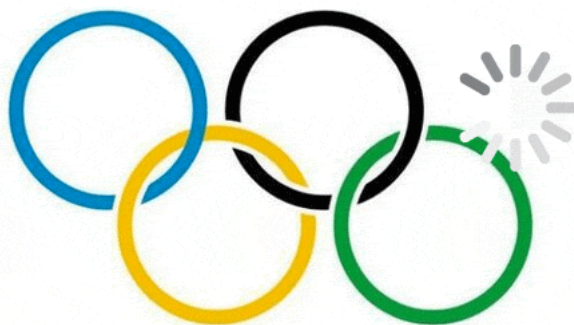
Async/await: 6 причин забыть о промисах

<https://hackernoon.com/6-reasons-why-javascripts-async-await-blows-promises-away-tutorial-c7ec10518dd9>

Node.JS, JavaScript, Блог компании RUVDS.com

Перевод

Если вы не в курсе, в Node.js, начиная с версии 7.6, встроена поддержка механизма `async/await`. Говорят о нём, конечно, уже давно, но одна когда для использования некоей функциональности нужны «костыли», и совсем другое, когда всё это идёт, что называется, «из коробки». Если ещё не пробовали `async/await` — обязательно попробуйте.



Сегодня мы рассмотрим шесть особенностей `async/await`, позволяющих отнести новый подход к написанию асинхронного кода к разряду инструментов, которые стоит освоить и использовать везде, где это возможно, заменив ими то, что было раньше.

Основы `async/await`

Для тех, кто не знаком с `async/await`, вот основные вещи, которые полезно будет знать прежде чем двигаться дальше.

- `Async/await` — это новый способ написания асинхронного кода. Раньше подобный код писали, пользуясь коллбэками и промисами.
- Наше предложение «забыть о промисах» не означает, что они потеряли актуальность в свете новой технологии. На самом деле, в основе `async/await` лежат промисы. Нужно учитывать, что этот механизм нельзя использовать с коллбэками.
- Конструкции, построенные с использованием `async/await`, как и промисы, не блокируют главный поток выполнения программы.
- Благодаря `async/await`, асинхронный код становится похожим на синхронный, да и в его поведении появляются черты такого кода, весь полезные в некоторых ситуациях, в которых промисами пользоваться было, по разным причинам, неудобно. Но раньше без них было не обойтись. Теперь же всё изменилось. Именно тут кроется мощь `async/await`.

Синтаксис

Представим, что у нас имеется функция `getJSON`, которая возвращает промис, при успешном разрешении которого возвращается JSON-объект. Мы хотим эту функцию вызвать, вывести в лог JSON-объект и вернуть `done`.

С использованием промисов подобное можно реализовать так:

```
const makeRequest = () =>
  getJSON()
  .then(data => {
    console.log(data)
    return "done"
  })

makeRequest()
```

Вот как то же самое делается с использованием `async/await`:

```
const makeRequest = async () => {
  console.log(await getJSON())
  return "done"
}

makeRequest()
```

Если сопоставить два вышеприведённых фрагмента кода, можно обнаружить следующее:

1. При описании функции использовано ключевое слово `async`. Ключевое слово `await` можно использовать только в функциях, определённых с использованием `async`. Любая подобная функция неявно возвращает промис, а значением, возвращённым при разрешении этого промиса, будет то, что возвратит инструкция `return`, в нашем случае это строка `done`.
2. Вышесказанное подразумевает, что ключевое слово `await` нельзя использовать вне `async`-функций, на верхнем уровне кода.

```
// Эта конструкция на верхнем уровне кода работать не будет
// await makeRequest()

// А такая - будет
makeRequest().then((result) => {
  // do something
})
```

3. Запись `await getJSON()` означает, что вызов `console.log` будет ожидать разрешения промиса `getJSON()`, после чего выведет то, что возвратит функция.

Почему `async/await` лучше промисов?

Рассмотрим обещанные шесть преимуществ `async/await` перед традиционными промисами.

1. Лаконичный и чистый код

Сравнивая два вышеприведённых примера, обратите внимание на то, насколько тот, где используется `async/await`, короче. И ведь речь, в данном случае, идёт о маленьких кусках кода, а если говорить о реальных программах, экономия будет ещё больше. Всё дело в том, что не нужно `.then`, создавать анонимную функцию для обработки ответа, или включать в код переменную с именем `data`, которая нам, по сути, не нужна. Кроме того, мы избавились от вложенных конструкций. Полезность этих мелких улучшений станет заметнее, когда мы рассмотрим другие примеры.

2. Обработка ошибок

Конструкция `async/await` наконец сделала возможной обработку синхронных и асинхронных ошибок с использованием одного и того же механизма — старого доброго `try/catch`. В следующем примере с промисами `try/catch` не обработает сбой, возникший при вызове `JSON`, так как он выполняется внутри промиса. Для обработки подобной ошибки нужно вызвать метод `.catch()` промиса и продублировать в нём обработку ошибок. В коде рабочего проекта обработка ошибок будет явно сложнее вызова `console.log` из примера.

```
const makeRequest = () => {
  try {
    getJSON()
      .then(result => {
```

```
// парсинг JSON может вызвать ошибку
const data = JSON.parse(result)
console.log(data)
})
// раскомментируйте этот блок для обработки асинхронных ошибок
// .catch((err) => {
//   console.log(err)
// })
} catch (err) {
  console.log(err)
}
```

Вот то же самое, переписанное с использованием `async/await`. Блок `catch` теперь будет обрабатывать и ошибки, возникшие при парсинге JS

```
const makeRequest = async () => {
  try {
    // парсинг JSON может вызвать ошибку
    const data = JSON.parse(await getJSON())
    console.log(data)
  } catch (err) {
    console.log(err)
  }
}
```

3. Проверка условий и выполнение асинхронных действий

Представьте, что надо написать кусок кода, который, загрузив некие данные, принимает решение о том, вернуть ли их в точку вызова, или, основываясь на том, что уже получено, запросить ещё что-нибудь. Решить подобную задачу можно так:

```
const makeRequest = () => {
  return getJSON()
  .then(data => {
    if (data.needsAnotherRequest) {
      return makeAnotherRequest(data)
      .then(moreData => {
        console.log(moreData)
        return moreData
      })
    } else {
      console.log(data)
      return data
    }
  })
}
```

Только от взгляда на эту конструкцию может разболеться голова. Очень легко потеряться во вложенных конструкциях (тут их 6 уровней), ск командах возврата, которые нужны лишь для того, чтобы доставить итоговый результат главному промису.

Код будет гораздо легче читать, если решить задачу с использованием `async/await`.

```
const makeRequest = async () => {
  const data = await getJSON()
  if (data.needsAnotherRequest) {
    const moreData = await makeAnotherRequest(data);
    console.log(moreData)
    return moreData
  } else {
    console.log(data)
    return data
  }
}
```

4. Промежуточные значения

задействуете оба результата от ранее вызванных промисов для вызова `promise3`. Вот как будет выглядеть код, решающий такую задачу.

```
const makeRequest = () => {
  return promise1()
    .then(value1 => {
      // do something
      return promise2(value1)
        .then(value2 => {
          // do something
          return promise3(value1, value2)
        })
    })
}
```

Если для `promise3` не нужно `value1`, можно без особых сложностей упростить структуру программы, особенно если вам режут глаз подобные конструкции, использованные без необходимости. В такой ситуации можно обернуть `value1` и `value2` в вызов `Promise.all` и избежать ненужных вложенных конструкций.

```
const makeRequest = () => {
  return promise1()
    .then(value1 => {
      // do something
      return Promise.all([value1, promise2(value1)])
    })
    .then(([value1, value2]) => {
      // do something
      return promise3(value1, value2)
    })
}
```

При таком подходе семантика приносится в жертву читабельности кода. Нет причин для того, чтобы помещать `value1` и `value2` в один и тот же массив за исключением того, чтобы избежать вложенности промисов.

То же самое можно написать с применением `async/await`, причём делается это удивительно просто, а то, что получается, оказывается интуитивно понятным. Тут поневоле задумаешься о том, сколько полезного можно сделать за то время, которое тратится на написание хоть сколько-нибудь приличного кода с использованием промисов.

```
const makeRequest = async () => {
  const value1 = await promise1()
  const value2 = await promise2(value1)
  return promise3(value1, value2)
}
```

5. Стеки ошибок

Представьте себе фрагмент кода, в котором имеется цепочка вызовов промисов, а где-то в этой цепочке выбрасывается ошибка.

```
const makeRequest = () => {
  return callAPromise()
    .then(() => callAPromise())
    .then(() => callAPromise())
    .then(() => callAPromise())
    .then(() => callAPromise())
    .then(() => {
      throw new Error("oops");
    })
}

makeRequest()
  .catch(err => {
    console.log(err);
    // вывод
    // Error: oops at callAPromise.then.then.then.then.then (index.js:8:13)
  })
```

Стек ошибки, возвращённый из цепочки промисов, не содержит указания на то, где именно произошла ошибка. Более того, сообщение об ошибке способно направить усилия по поиску проблемы по ложному пути. Единственное имя функции, которое содержится в сообщении — `callAPromise`, а эта функция никаких ошибок не вызывает (хотя, конечно, тут есть и полезная информация — сведения о файле, где произошла ошибка, и о номере строки).

Если же взглянуть на подобную ситуацию при использовании `async/await`, стек ошибки укажет на ту функцию, в которой возникла проблема

```
const makeRequest = async () => {
  await callAPromise()
  await callAPromise()
  await callAPromise()
  await callAPromise()
  await callAPromise()
  throw new Error("oops");
}

makeRequest()
  .catch(err => {
    console.log(err);
    // ВЫВОД
    // Error: oops at makeRequest (index.js:7:9)
  })
```

Подобное нельзя назвать огромным плюсом, если речь идёт о разработке в локальном окружении, когда файл с кодом открыт в редакторе — оказывается весьма полезным, если вы пытаетесь понять причину ошибки, анализируя лог-файл, полученный с продакшн-сервера. В подобных случаях знать, что ошибка произошла в `makeRequest`, лучше, чем знать, что источник ошибки — некий `then`, вызванный после ещё одного `then`, который следует за ещё каким-то `then`...

6. Отладка

Этот пункт последний, но это не значит, что он не особо важен. Ценнейшая особенность использования `async/await` заключается в том, что с его помощью задействована эта конструкция, гораздо легче отлаживать. Отладка промисов всегда была кошмаром по двум причинам.

1. Нельзя установить точку останова в стрелочных функциях, которые возвращают результаты выполнения выражений (нет тела функции).

```
4
5   const makeRequest = () => {
6     return callAPromise()
7       .then(() => callAPromise())
8       .then(() => callAPromise())
9       .then(() => callAPromise())
10      .then(() => callAPromise())
11   }
12
```

Попробуйте поставить где-нибудь в этом коде точку останова

2. Если вы установите точку останова внутри блока `.then` и воспользуетесь командой отладчика вроде «шаг с обходом» (`step-over`), отладчик перейдёт к следующему `.then`, так как он может «перешагивать» только через синхронные блоки кода.

При использовании `async/await` особой нужды в стрелочных функциях нет, и можно «шагать» по вызовам, выполненным с ключевым словом `await`, так же, как это делается при обычных синхронных вызовах.

```

4
5  const makeRequest = async () => {
6    await callAPromise()
7    await callAPromise()
8    await callAPromise()
9    await callAPromise()
10   await callAPromise()
11  }
12

```

Отладка при использовании `async/await`

Замечания

Вполне возможно, у вас возникнут некоторые соображения не в пользу применения `async/await`, продиктованные здоровым скептицизмом. Прокомментируем пару наиболее вероятных.

- **Эта конструкция делает менее очевидными асинхронные вызовы.** Да, это так, и тот, кто привык видеть асинхронность там, где есть коллбэк или `.then`, может потратить несколько недель на то, чтобы перестроиться на автоматическое восприятие инструкций `async/await`. Однако, например, в C# подобная функциональность есть уже многие годы, и те, кто с этим знакомы, знают, что польза от неё стоит временных неудобств при чтении кода.
- **Node 7 — не LTS-релиз.** Это так, но совсем скоро выйдет Node 8, и перевод кода на новую версию, вероятно, не потребует особых усилий.


Выводы


Пожалуй, `async/await` — это одна из самых полезных революционных возможностей, добавленных в JavaScript в последние несколько лет. даёт простые и удобные способы написания и отладки асинхронного кода. Полагаем, `async/await` пригодится всем, кому приходится писать код.

Уважаемые читатели! Как вы относитесь к `async/await`? Пользуетесь ли вы этой возможностью в своих проектах?

Метки: JavaScript, `async/await`, промисы, Node.js

↑ +41 ↓ 200 36,1k 182


RUVDS.com 681,62
 RUVDS – хостинг VDS/VPS серверов


53,0 Карма **407,2** Рейтинг **76** Подписчики
 @ru_vds
 Пользователь

[Facebook](#) [Twitter](#) [Вконтакте](#) [Google+](#)

Поделиться публикацией

ПОХОЖИЕ ПУБЛИКАЦИИ

7 июля в 13:57

Платформа Node.js обойдёт Java в течение года

↑ +14 23,3k 47 56

30 мая в 15:33

Node.js и cote: простая и удобная разработка микросервисов

↑ +18

👁 13k

📖 133

💬 15

3 февраля в 15:07

Node.js, Express и MongoDB: API за полчаса

↑ +21

👁 55k

📖 312

💬 26

Комментарии 182



k12th 10.04.17 в 15:31

📖

↑

Мне кажется, что комитет слишком увлекся синтаксическими плюшками в ущерб стандартной библиотеке. async/await это здорово, конечно, но для вменяемой работы с датами надо до сих пор ставить сторонний пакет, а ведь это гораздо проще стандартизировать и полифиллить, чем новый синт. Проблемы с left-pad можно было легко избежать, если бы этот метод был в стандартной библиотеке.



Jaromir 10.04.17 в 15:49

📖 ↻ 🔍

↑

> Мне кажется, что комитет слишком увлекся синтаксическими плюшками
Нет. На ES6+ писать намного приятней, чем на ES5. Порог вхождения, правда, существенно выше

> Проблемы с left-pad можно было легко избежать, если бы этот метод был в стандартной библиотеке
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/padStart



k12th 10.04.17 в 15:56

📖 ↻ 🔍

↑

На ES6+ писать намного приятней, чем на ES5.

Кто спорит?

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/padStart

Это можно и нужно было сделать еще в ES5, а попадет это только в ES2017.



monolithed 10.04.17 в 15:49

📖 ↻ 🔍

↑

Для работы с датами есть Intl.DateTimeFormat и Date.prototype.toLocaleString. Базовые потребности они удовлетворяют более чем.



k12th 10.04.17 в 15:59

📖 ↻ 🔍

↑

Да? И как там вывести дату в формате DD.MM.YYYY?



spmbt 10.04.17 в 17:59

📖 ↻ 🔍

↑

Очень просто,

```
Date().replace(/(.+?) (\w{3}) (\d+) (\d{4}) (.+)/, '$3.$2.$4')
```

ну и там строчку в число перевести массивом немного:)



Mingun 10.04.17 в 20:41

📖 ↻ 🔍

↑

Это называется не «очень просто», а «жуткий костыль». Вот просто:

```
moment().format('DD.MM.YYYY');
```



mayorgovp 10.04.17 в 20:42

📖 ↻ 🔍

↑

Сарказм-детектор проверьте, кажется он у вас поломался



monolithed 13.04.17 в 16:28

📖 ↻ 🔍

↑

```
let date = new Date();

date.toLocaleString('ru-RU', { day: 'numeric', month: 'numeric', year: 'numeric' });
```

 k12th 13.04.17 в 16:30 # 📌 🔄

Офигенное API. Но вроде бы работает, справедливости ради.

 monolithed 14.04.17 в 20:24 # 📌 🔄

Можно еще короче:

```
new Intl.DateTimeFormat('ru').format(new Date());
```

 spmbt 15.04.17 в 14:08 # 📌 🔄

И ещё: Intl.DateTimeFormat().format(new Date)

Да, и вот как без неё:

```
var dateFormat = () => Date().replace(/.+?(\w\w) (\d+) (\d+) .+/, (a, i, j, k) => j + '.' +  
    + ('anebarprayunulugepctovec'.indexOf(i) / 2 + 101 + '').substr(1) + '.' + k);
```

 mayorovp 10.04.17 в 18:10 # 📌 🔄

Видите ли в чем дело — поставить сторонний пакет проще чем "поставить" синтаксическую плюшку. Поэтому синтаксические плюшки — это в язь самое важное.

 kahi4 10.04.17 в 18:35 # 📌 🔄

Для работы с датами стандартом de-facto стал moment.js, несмотря на корявость своего API (особенно его любовь мутировать даты). Меня уже д мучает вопрос, почему просто не взять и не включить его как стандарт? Доработать API и красота.

 k12th 10.04.17 в 19:09 # 📌 🔄

Не уверен, что одобряю эту идею. Помнится, были предложения включать jQuery прямо в состав браузера — мол, все равно 95% сайтов его использует. Никто не стал этого делать, и слава богу.

 arvitaly 10.04.17 в 20:16 # 📌 🔄

Ну главную часть все же добавили, querySelector, даже переменная в некоторых браузерах та же (\$).

 TheShock 10.04.17 в 21:35 # 📌 🔄

даже переменная в некоторых браузерах та же (\$).

Разве?

 mayorovp 10.04.17 в 21:37 # 📌 🔄

Видимо, имелась в виду вспомогательная функция, видимая в инструментах разработчика.

 oWeRQ 11.04.17 в 16:30 # 📌 🔄

А также classList, map, each, Promises, formData и fetch, css анимации, да в общем-то почти все в том или ином виде застандартизировал слава богу, что не под копирку, только про цепочки вызовов забыли.

 comerc 10.04.17 в 20:49 # 📌 🔄

Возьмите date-fns :)

 raveclassic 10.04.17 в 23:18 # 📌 🔄

Только предложить хотел :)

 dreammaster19 10.04.17 в 15:50 # 📌

В конструкции then не надо оборачивать в новый промис с then, поскольку return возвращает промис, надо просто продолжить цепочку. async/await конечно криво и удобно но не дает какой-то панацеи и в некоторых случаях промисы очень удобны. Взять хотябы Promise.all(). Так что не понимаю зачем делать категоричные заявления об отказах от текущего функционала, если можно просто всё использовать в нужных мустах

 Razaz 10.04.17 в 16:25 # 📌 🔄

К слову в C# есть await Task.WhenAll(awaitable1, awaitable2). Как я понимаю если Promise.all() возвращает промис, То можно сделать то же самое



Quilin 10.04.17 в 16:42



Определенно можно. Когда в C# появился синтаксис `async/await`, у меня лично весь асинхронный код стал гораздо симпатичнее и понятнее. Дичайше обрадовался тому же синтаксису в JS, они молодцы что выбрали именно C# в качестве примера для подражания.



Razaz 10.04.17 в 16:51



Спасибо. Только начинаем слезать с ES5 и еще не успел все пощупать :)



dreammaster19 10.04.17 в 17:46



Я к тому, что нет `await all`, да и `async` это лишь синтаксический сахар для промисов, так что про них не стоит забывать. Но вот на счет исключения `async/await` я полностью согласен с автором, гораздо удобнее



mayorovp 10.04.17 в 18:04



Зачем нужен `await all` если можно написать `await Promise.all(...)`?



lam0x86 11.04.17 в 00:48



Изначально, кстати, предлагался вариант:

```
async function concurrent () {
  var [r1, r2, r3] = await* [p1, p2, p3];
}
```

... но он был отклонён.



Pongo 10.04.17 в 18:56



| `async` это лишь синтаксический сахар для промисов

Джаваскрипт по-особому обрабатывает `await` в циклах и условных операторах — с промисами такого поведения нет — поэтому это нечто большее, чем синтаксический сахар.



raveclassic 10.04.17 в 23:31



| `async` это лишь синтаксический сахар для промисов

| нечто большее, чем синтаксический сахар.

Я в кишках V8 не копался, но у меня есть стойкое подозрение, что `async/await` работает поверх генераторов с резолвом `yield`-енного промиса при прогоне результирующего итератора. Результат `.catch` же скармливается `.throw`. Слишком уж «гибкий» для промисов фло получается, на них такого без бубнов не сделать, взять те же циклы.

Посмотрите, во что бабель транспайлит `async/await` с включенным `transform-async-to-generator`. Ну *очень* похоже, что V8 делает то же самое при чем на том же JS, как в случае с промисами.



Sirion 11.04.17 в 01:08



Если я перечитал этот коммент два раза и всё равно не понял, мне ещё можно быть веб-девелопером или пора уходить в домохозяйк



raveclassic 11.04.17 в 10:05



Постараюсь подробнее. Возьмем небольшой пример, одобренный `async/await`:

```
//исходник с разными вариантами async/await
const delay = ms => new Promise(resolve => {
  setTimeout(() => resolve(ms), ms);
});

const throwAsync = (error, ms) => new Promise((resolve, reject) => {
  setTimeout(() => reject(new Error(error)), ms);
});

const foo = async (n, shouldThrow) => {
  for (let i = 1; i < n + 1; i++) {
    const result = await delay(i * 1000);
    console.log('wait', result);
  }

  if (shouldThrow) {
    await throwAsync('Should throw', 1000);
  }

  return 'done';
}
```

```

}

foo(3).then(console.log).catch(console.error);
foo(3, true).then(console.log).catch(console.error);

```

```

//результат после бабеля с transform-async-to-generator
var _context;

function _asyncToGenerator(fn) {
  return function() {
    var gen = fn.apply(this, arguments);
    return new Promise(function(resolve, reject) {
      function step(key, arg) {
        try {
          var info = gen[key](arg);
          var value = info.value;
        } catch (error) {
          reject(error);
          return;
        }
        if (info.done) {
          resolve(value);
        } else {
          return Promise.resolve(value).then(function(value) {
            step("next", value);
          }, function(err) {
            step("throw", err);
          });
        }
      }
      return step("next");
    });
  };
}

const delay = ms => new Promise(resolve => {
  setTimeout(() => resolve(ms), ms);
});

const throwAsync = (error, ms) => new Promise((resolve, reject) => {
  setTimeout(() => reject(new Error(error)), ms);
});

const foo = () => {
  var _ref = _asyncToGenerator(function*(n, shouldThrow) {
    for (let i = 1; i < n + 1; i++) {
      const result = yield delay(i * 1000);
      console.log('wait', result);
    }

    if (shouldThrow) {
      yield throwAsync('Should throw', 1000);
    }

    return 'done';
  });

  return function foo(_x, _x2) {
    return _ref.apply(this, arguments);
  };
}();

foo(3).then((_context = console).log.bind(_context)).catch((_context = console).error.bind(_context));
foo(3, true).then((_context = console).log.bind(_context)).catch((_context = console).error.bind(_context));

```

Уделите внимание функции `_asyncToGenerator`, как она работает с промисами, принимаемыми из `step("next")`. Еще посмотрите все `await` заменились на `yield` абсолютно в тех же местах без перекомпоновки кода, что пришлось бы делать при переписывании промисы.

Исходный посыл был к тому, что есть очень большие подозрения, что `async/await` внутри V8 сделан не на основе промисов, а как р: основе генераторов.

Поиграться можно вот с таким набором пресетов.



mayorovp 11.04.17 в 10:12

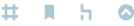


Исходный посыл был к тому, что есть очень большие подозрения, что `async/await` внутри V8 сделан не на основе промисов, а на основе генераторов.

Почему вы разделяете эти два случая?



raveclassic 11.04.17 в 10:22



Не разделяю, а дополняю следующее:

`async` это лишь синтаксический сахар для промисов

Действительно, промисы и генераторы работают в тандеме в случае `async/await`. `Async` функция — это возвращаемый промис `await` точки — `yield` в генераторе.



raveclassic 11.04.17 в 10:24



Отсюда и древний холивар, что `async/await` не нужен, так как все то же самое можно сделать на генераторах и маленькой функции-хелпере.



iShatokhin 11.04.17 в 15:26



Да, но зато можно определить через символы является ли функция `async`:

```
var fn = async function () {};  
fn[Symbol.toStringTag] === "AsyncFunction";
```



bano-notit 11.04.17 в 16:41



Ну и в чём прикол? В том что они реализованы на уровне движка js? Какая разница, они все реализованы одинаково по смыслу, просто так они реализованы в движке, а так их нужно реализовывать самим.



raveclassic 11.04.17 в 17:12



И зачем вам это, скажите, пожалуйста?

Через символы можно много чего определить, в том числе и генераторы:

```
(function * foo() {})[Symbol.toStringTag] === 'GeneratorFunction'
```

Другое дело, что смысловой нагрузки это никакой не несет.



iShatokhin 11.04.17 в 17:13



Пример использования — <https://github.com/caolan/async/pull/1390>



raveclassic 11.04.17 в 17:30



Это все, конечно, прекрасно, только из вашего примера вытекает только то, что `async/await` существует для того чтобы его можно было задектить через `Symbol.toStringTag`. Генераторы тоже детектятся, а решают те же задачи даже чуть больше.

И если уж вы используете стороннюю библиотеку, то хелпер для «размотки» генератора есть в `co`.



iShatokhin 11.04.17 в 17:35



Я не пользуюсь именно этой библиотекой, просто пример, где это используется. Сам я буду использовать "AsyncFunction" немного по другому, для определения функций в обертке и отслеживании вызовов API внутри системы (аналитика производительности, мониторинг ошибок).



mayorovp 11.04.17 в 08:34



Нет никакого противоречия. Когда метод в конечный автомат или генератор превращает транслятор — это нормально. Когда то же сам делает программист вручную — это пляски с бубном и костылями.



raveclassic 11.04.17 в 10:02



Нет никакого противоречия.

Хм, я не имел в виду противоречие про сахар, а просто объединил =)



unel 11.04.17 в 22:37

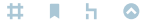


Я бы не стал утверждать, что любой конечный автомат, реализованный программистом — это пляски с бубном и костылями.

Если это требуется для реализации задачи — то почему нет?



DistortNeo 11.04.17 в 14:43



Не знаю, как в JS, а в C# за асинхронными функциями стоит довольно суровая кодогенерация. Грубо говоря, каждая функция разворачивается в класс, переменные внутри функции — в поля класса, а ветки выполнения, содержащие await — либо в отдельные функции, либо в функции с большим switch (можно и так, и так реализовывать машину состояний).

Можно этот класс написать вручную, тогда код с использованием async/await будет идентичен промисам. Так что async/await — это именно сахар.



raveclassic 11.04.17 в 14:47



async/await раскладывается бабелем при транспайлинге в es5 как раз в свитч вместе с regeneratorRuntime (рантайм который этот свитч перемалывает). Собственно, генераторы раскладываются в похожий свитч с тем же рантаймом.



Quilin 10.04.17 в 19:19



К сожалению, не очень хорошо разбираюсь в подкапотной работе V8 в оптимизации async/await, но немножко могу про C#, и там компилятор очень умеет оптимизировать код. Скажем, когда результаты функций должны использоваться совместно:

```
let a = await _a()
let b = await _b()
let c = await _c(a, b)
console.log(c)
```

В мире с промисами это будет выглядеть как-то вот так:

```
Promise.all(_a(), _b())
  .then(r => _c(r[0], r[1]))
  .then(console.log)
```

Может быть и ничего так, но гораздо сложнее для чтения и понимания, а если между строками ставить еще какие-нибудь логи или вызовы синхронных функций, то код на промисах станет гораздо сложнее.

В C# эти страшные «промисы» (Task) резолвит внутренний оптимизатор, по сути оно во что-то такое и превращается, но пользоваться этим гораздо удобнее, имхо.



Apathetic 10.04.17 в 20:45



Нет. Ваш пример с await с обычными промисами выглядел бы так:

```
_a()
  .then(a => _b().then(b => _c(a, b)))
  .then(console.log);
```

То есть еще хуже, чем вы написали. Сначала выполнится промис _a, и только потом — _b.

Чтобы промисы работали синхронно, действительно нужно использовать Promise.all:

```
let [a, b] = await Promise.all([_a(), _b()]);
let c = await _c(a, b);
console.log(c);
```



mayorovp 10.04.17 в 21:39



Необязательно. Можно же и вот так сделать:

```
let a = _a();
let b = _b();
let c = _c(await a, await b);
console.log(await c);
```

Но с тем, что первый пример был некорректен — согласен.

 **Apathetic** 10.04.17 в 21:56 # 1 2 3 4

Нельзя. Сделаем такой простенький промис для демонстрации:

```
const p = (a, b) => new Promise(resolve => setTimeout(resolve, 1000, b ? { a, b } : a));
```

Вопрос на засыпку: когда в консоли появится результат — через секунду, две или три?

```
(async () => {
  console.log(await p(await p(1), await p(2)));
})();
```

Или более развернуто (как у вас):


```
(async () => {
  let a = p(1);
  let b = p(2);
  let c = p(await a, await b);
  console.log(await c);
})();
```

 **mayorovp** 10.04.17 в 21:58 # 1 2 3 4

Вы изменили мой код! Он должен выглядеть вот так:

```
(async () => {
  let a = p(1);
  let b = p(2);
  console.log(await p(await a, await b));
})();
```

Результат в консоли появится через 2 секунды. А приведенный вами код покажет его через 3 секунды.

 **Apathetic** 10.04.17 в 22:10 # 1 2 3 4

Вообще, развернутый вариант у меня в комментарии тоже за 2 секунды отработает.

Причины такого поведения понятны, но совершенно (ИМХО) не очевидны. Попробую использовать вопрос о разнице между двумя кусками кода и её причинах на собеседованиях =)

 **mayorovp** 10.04.17 в 22:12 # 1 2 3 4

Они очевидны и интуитивно понятны любому, кто писал асинхронные программы на C# используя Begin/End пары методов

 **Apathetic** 10.04.17 в 22:13 # 1 2 3 4

У меня такие не часто попадают.

 **Apathetic** 10.04.17 в 22:13 # 1 2 3 4

То есть получается, там, где я сказал "нельзя", на самом деле можно. Довольно-таки глупо с моей стороны не заметить, что о промиса (а и b) запускаются сразу друг за другом.

 **Quilin** 11.04.17 в 16:36 # 1 2 3 4

Да, вы правы, я действительно ошибся. Но тем не менее аргумент мой остается тем же самым — код с промисами гораздо сложнее дчтения и обрастает морем ненужных скобок, если нам нужно логирование или дополнительные действия.

 **Apathetic** 11.04.17 в 16:39 # 1 2 3 4

Это да

 **HaMI** 10.04.17 в 16:11 # 1

Всю статью можно свести к этому предложению

>Однако, например, в С# подобная функциональность есть уже многие годы, и те, кто с этим знакомы, знают, что польза от неё стоит временных неудобств при чтении кода.

Я не луддит мне просто не очевидно зачем менять один синтаксис на другой.

Второе, все приведенные «плюшки» совсем меркнут если использовать bluebird – он очень улучшает работу с ошибками и дает кучу утилит для координации промисов.



Nakosika 10.04.17 в 16:32 # 📌 📄 🔄



Императивный код проще чем каллбэк спагетти в любое время дня и ночи. Движение языка в сторону упрощения это офигеть как круто. Обычно все только усложняются...



HaMI 10.04.17 в 17:30 # 📌 📄 🔄



промахнулся комментарием – ответил вам ниже



HaMI 10.04.17 в 17:29 # 📌



стоп, кто говорил про callback спагетти? предлагают забыть о промисах

тут предлагают использовать async/await и забыть о промисах делая чуть короче простые примеры. Но ничего не сказано про более сложные кейсы делать с типизацией ошибок? как координировать несколько промисов? все это решает bluebird

второе, статья манипулятивна:

```
const makeRequest = () => {
  return promisel()
    .then(value1 => {
      // do something
      return Promise.all([value1, promise2(value1)])
    })
    .then([value1, value2] => {
      // do something
      return promise3(value1, value2)
    })
}
```

так выглядит намного короче

```
const makeRequest = () => promisel()
  .then(value1 => Promise.all([value1, promise2(value1)]))
  .then([value1, value2] => promise3(value1, value2))
```

а с блюбердом совсем так:

```
const makeRequest = () => promisel()
  .then(value1 => [value1, promise2(value1)])
  .spread([value1, value2] => promise3(value1, value2))
```

>Движение языка в сторону упрощения

js и так простой, то что с этой «фичей» асинхронный код начинает выглядеть как синхронных – сомнительный плюс.

код ниже не имеет смысла – с тем же успехом можно писать синхронно и не заморачиваться и есть вероятность что даже быстрее будет

```
const makeRequest = () => {
  return callAPromise()
    .then(() => callAPromise())
    .then(() => callAPromise())
    .then(() => callAPromise())
    .then(() => callAPromise())
}
```



Nakosika 10.04.17 в 17:37 # 📌 📄 🔄



Ну в статье довольно понятно какие плюсы. Из каллбеков к примеру ретурн не сделать или трайкатч, потому что вся информация о стеке уже буд потеряна.

 **Nakosika** 10.04.17 в 17:43    

Я сам по началу не разобрался и думал нафига козе баян, а код не просто выглядит как синхронный, он им и является, потому что стек сохраняет отсюда все плюсы.

 **HaMI** 10.04.17 в 17:54    

у меня сложилось впечатление что вы путаете callback и promise.

>он им и является, потому что стек сохраняется отсюда все плюсы.

вы уверены в своих словах? в чем профит? юзайте все что *sync из стандартной библиотеки – будет вам счастье.

 **Nakosika** 10.04.17 в 18:09    





В пропозишен вроде так написано, но сам я еще палкой не тыкал туда. Надо будет проверить что у них вышло.

 **spmbt** 10.04.17 в 17:47    

Да, коллбеки (промисы) с исключениями — самая большая у них проблема. Раньше просто async не было, поэтому предлагали меньшее зло, тем более, что того же синтаксиса, но упорно шли к async и чуть раньше — к урожаю:).

 **mayorovp** 10.04.17 в 18:07    




Вам никто не мешает использовать async/await и bluebird вместе.

 **HaMI** 10.04.17 в 18:18    

конечно, но статья называется «Async/await: 6 причин забыть о промисах». Что само по себе уже смешно – потому как предлагают забыть .then и .catch методах промисов.

 **Deosis** 11.04.17 в 07:55    

Вот только вы не учли, что под // do something может оказаться страница кода, которую вы смело выкинули.

 **HaMI** 11.04.17 в 11:39    


ага, вам не показалось странным что в примере с Promise «do something» есть, а в примере с async/await нет?

или вы думаете что «do something» во втором случае будет меньше?

 **HaMI** 10.04.17 в 18:02  

Если располагать тоже количество «логики» на строчку кода, то выйдет что примеры аналогичны по длине
Например:

```
const makeRequest = async () => {
  console.log(await getJSON())
  return "done"
}
//против:
const makeRequest = () =>
  getJSON().then(console.log)
  .then(()=> 'done')
```

 **Claud** 10.04.17 в 18:35  

```
async () => {
  let t1 = await call();
  let t2 = await call();
  let t3 = await call();
}
```



Я верно понимаю, что в примере выше call() будет вызываться асинхронно, и не дожидаться завершения предыдущих вызовов?

 **mayorovp** 10.04.17 в 19:00    

Нет. Каждый await ожидает завершения прежде чем передать управление дальше.

Чтобы было как вы написали — надо делать так (пишу по памяти, могу напутать с синтаксисом немного):

```
var [f1, f2, f3] = await Promise.all([call(), call(), call()])
```

 **HaMI** 10.04.17 в 19:38    

тут важно добавить — что call таки может быть «асинхронным»(например вызвать что-то по сети) и не будет блокировать event-loop(а значит где другом месте что-то может выполниться)

Но сам кусок написан так что call'ы будут выполняться по очень — если нужно условно одновременно, то нужен Promise.all как mayorovp написал

 **Maxxon** 18.04.17 в 01:22    

как вариант без Promise.all

```
async () => {  
  let t1 = call();  
  let t2 = call();  
  let t3 = call();  
  
  let f1 = await t1;  
  let f2 = await t2;  
  let f3 = await t3;  
}
```

 **HaMI** 18.04.17 в 12:42    

Что вы хотели показать своим примером?

 **Maxxon** 18.04.17 в 13:17    

Очевидно, что то что и написал, код выше эквивалентен этому

```
var [f1, f2, f3] = await Promise.all([call(), call(), call()])
```

 **iShatokhin** 18.04.17 в 19:12    

Не эквивалентен, т.к. try-catch с таким кодом не работает, реально будет поймано исключение только для первого await, остальные уйдут "Unhandled promise rejection" (с Promise.all такой проблемы не будет). В соседнем топике обсуждалось — https://habrahabr.ru/post/326442/#comment_10177554

 **Maxxon** 18.04.17 в 21:30    

Ну так вроде ничто не мешает каждый await обернуть в try-catch, за одно удобнее различные исключения бросать

 **bano-notit**  18.04.17 в 17:46    



Последовательно, опять же... А нужно именно паралельное. Допустим t1 будет идти 7 мс, t2 — 3 мс, t3 — 12 мс. Если сделать через Promise.all(), то даже если мы будем учитывать, что на вызов call уходит само по себе 1 мс, получится всего $\max(12, 7, 3) + 3 = 15$ мс, в отличие от ваших последовательных $7 + 1 + 3 + 1 + 12 + 1 = 25$ мс. Мне кажется, что преимущество Promise.all() достаточно очевидно.

 **HaMI**  18.04.17 в 18:15    

@bano-notit

вы не поверите, но таки представленный @maxxon код «почти»(с оговорками для ноды) паралельный и медленнее того который с Promise на ~2ms — и то, есть вероятность, что это лаг, а не реальное замедление

Другой вопрос — что это не очевидно, и синтаксически его код не показывает что он «паралельный»

 **bano-notit** 18.04.17 в 18:23    

Блин... Согласен, не досмотрел. Ведь промисы все просто в переменных сохраняются, а не генерятся на самом await...

Но то, что это совершенно не очевидно — 100%.

 **Maxxon** 18.04.17 в 21:32    

await и генерит промисы, как видно из названия он их ждет



 **TheShock** 18.04.17 в 18:25    

Другой вопрос – что это не очевидно, и синтаксически его код не показывает что он «параллельный»



`call()` — запуск запроса, а `await` — ожидания его выполнения. То есть сначала запускается три запроса и только потом запускается и ожидание. Мне было вполне очевидно, что они исполняются параллельно. Видимо, необходимо базово понимать, как этот код работает

 **mayorovp** 19.04.17 в 07:48    

Ну, тут проблема не столько в операторе `await`, сколько в функции `call`. По названию не видно что она асинхронная.



 **comerc** 10.04.17 в 20:58  

В статье ни слова про это: <https://habrahabr.ru/post/320306/>



 **TheShock** 10.04.17 в 21:41    

А зачем вспоминать про статью демагога, которая может вызвать только фейспалм?



 **comerc** 15.04.17 в 14:29    


У меня вызывают подобную реакцию ваши оценочные суждения.



 **TheShock** 15.04.17 в 15:57    

Ваше сообщение сообщение имело б смысл. Если бы там в комментариях не было аргументов того, что статья фиговая.



 **bano-notit**  11.04.17 в 00:39  

Вопрос такой: как с помощью `async/await` организовать тот же самый `Promise.all()`? Ведь в таком коде



```
let a = await first(),
    b = await second();
```

Второй промис будет выполнен только после того, как выполнится первый. А ведь второй от первого не зависит, так что такой вот код сделать нормальным можно только вот так вот:

```
let [a, b] = await Promise.all([first(), second()]).then((a, b) => [a, b]);
```

А это как-то не сильно входит в идею синтаксического сахарочка...

 **bano-notit** 11.04.17 в 00:45    

Как показали тесты `.then()` в данном случае не является нужным... Но всё равно конструкция получается не сильно сладкая.



 **Akuma** 11.04.17 в 00:52    

Вот тут ответили:

https://habrahabr.ru/company/ruvds/blog/326074/#comment_10164346



Но, честно говоря, с `Promise.all` как-то интуитивней становится. А то слишком замороченно получается.

 **bano-notit** 11.04.17 в 00:54    

Да вообще-то можно было бы вообще для всего промис интерфейса сделать сахарок, например логические операторы для промисов) Это бы вообще офигительно нечитаемо.



 **TheShock** 11.04.17 в 02:31    

ну не знаю, по-моему вполне читаемо:



```
let [a, b] = await [ first(), second() ];
```

 **bano-notit** 11.04.17 в 02:36    



Не согласен. потому что по идее эта конструкция должна выглядеть так:

```
let [a, b] = [await first(), await second()];
```

Ведь массив не является промисом... А в качестве синтаксического сахара принимать массивы... Ну это получается нужно ещё и такую вещь реализовывать:

```
let {first: a, second: b} = await {first: first(), second: second()}
```

Да и что тогда делать, например с такой вещью?

```
let [a, b] = await [1, second()]
```

Ведь такое вполне может произойти... `Promise.all()` с этим справляется, но вот реализовать такое на синтаксическом сахаре будет уж много менее явно и понятно со стороны.



TheShock 11.04.17 в 02:39 # 📌 📄 🔄



```
let {first: a, second: b} = await {first: first(), second: second()}
```

Это валидно, остальное пусть валится с ошибкой.



bano-notit 11.04.17 в 02:41 # 📌 📄 🔄



Хорошо так придумано! То есть `Promise.all()` может, а синтаксический сахар к нему пусть с ошибкой валится...



TheShock 11.04.17 в 02:45 # 📌 📄 🔄



Да, тут вы правы. Интерфейс стоит как у `all` сделать.



bano-notit 11.04.17 в 02:48 # 📌 📄 🔄



Тогда без объектов... Тогда не весь изначальный синтаксический сахар присвоения значений получается... Короче говоря диг



TheShock 11.04.17 в 02:58 # 📌 📄 🔄



JS — вечная проблема с тем, что новые фишки спотыкаются о старые костыли. Я б возможности `.all` добавил, но не убирал логически ожидаемое поведение с объектами.



bano-notit 11.04.17 в 03:04 # 📌 📄 🔄



Понимаете ли, в этом то и прикол, что нужно либо добавлять объекты внутрь `Promise.all()`, что не сильно влечёт за с проблемы, либо не делать этой прикольной штуки для объектов, либо сделать ещё умнее, сделать `await` отдельным оператором, который принимает только промисы, а остальные значения возвращает через `Promise.resolve()`, как собственно и сделали они)



TheShock 11.04.17 в 03:17 # 📌 📄 🔄



Не могу согласиться, что умнее. Промисы — костыль своего времени из-за отсутствия евейта. Теперь мы продолжаем ставить костыли, чтобы соответствовать старым костылям и из-за этого недостаточно развиваемся.

Да, должна быть обратная совместимость, т.е. код, который раньше использовал промисы легко переключается на э... но это не значит, что мы должны слепо следовать устаревшим контрактам.



bano-notit 11.04.17 в 03:31 # 📌 📄 🔄



На счёт поддержки старого у меня почему-то вспоминается сразу ie... Не знаю почему, но ассоциация у меня прям жёсткая, когда слышу/читаю слово "совместимость"...

А на счёт того, что должны или нет, тк вопрос в том, что js сам по себе начинает поддерживаться по кускам, поэтом сейчас есть смысл вводить только сахарок, который можно будет переделать на старые версии. Тут это логично, п что есть до сих пор люди, которые сидят на ie7-9, которым про наши холивары "что лучше, Promise или await/async вообще пофигу, у них не работает ни то, ни другое по дефолту.



TheShock 11.04.17 в 03:35 # 📌 📄 🔄



У них `async-await` и так и так не работает, потому можно сделать нормальный интерфейс.



bano-notit 11.04.17 в 03:49 # 📌 📄 🔄



Я именно про этом, потому что там нужно, получается, 2 раза фигачить: 1 — `async/await` трансформировать промисы; 2 — взять и подключить отдельную либу для того, чтобы хотя бы не нативные, но промисы были.



mayorovp 11.04.17 в 08:39



Промисы — костыль своего времени из-за отсутствия евейта.

Нет, это не так. Невозможно сразу сделать оператор `await`, не вводя в язык промисы или их аналоги.



bano-notit 11.04.17 в 03:06



На счёт `as` они просто попытались спереть у питона... Но у них не получилось, потому что модули в `js` всё равно редатировать труднее, чем питоновские, там потому что имя модуля, из которого всё берётся находится раньше вещей, которые из него берутся, что в `insert mode` ещё хоть как-то можно понять. А вот когда тебе кроме того, что изменить `имп` нужно ещё и путь к модулю запоминать, вот тут уже начинается портативный ад.



TheShock 11.04.17 в 03:21



Лучше бы они раньше взяли бы пример с питона и сделали нормальную деструктуризацию. А импорты да — крайне кривые.



unel 11.04.17 в 13:11



Глядя на некоторые из этих примеров использования промисов и правда может разболеться голова, ведь промисы как раз таки и были задуманы для решения `callback-hell` и этого адского уровня вложенности.

Но некоторые из них же можно переписать!
например, вместо

этого кода:

я бы написал

попроще:

а вместо

вложенных промежуточных переменных

сделал бы

плоский список:

но с другой стороны, не могу не согласиться, что с `async/await` это выглядит поопрятней =)



Shannon 11.04.17 в 15:48



Можно-то можно, даже `callback-hell` можно переписать и превратить во что-то приятное и понятное

Но вот такое:

```
async function gogogo(queryList) {
  let resultJson = {}

  for (const query of queryList) {
    let json = {}
    const type = await getType(query)

    if (type === 1) {
      json = await getJson(query)
    }
    else if (type === 2) {
      const rawData = await getRawData(query)
      json = await rawToJson(rawData)
    }
    else {
      break
    }

    Object.assign(resultJson, json)
  }

  return resultJson
}
```

```

async function doSomething() {
  const resultJson = await gogogo(queryList)
  const html = templateGenerator(resultJson)
  console.log(hmtl)
}

doSomething()

```

Уже становится немного сложновато представить на промисах, особенно если какой-то еще логики надо добавить

 HaMI 11.04.17 в 17:42    

Сорри, по понятным причинам не тестал. Попрошу обратить еще внимание на <http://bluebirdjs.com/docs/api/promise.map.html#map-option-concurrency> — что-то мне подсказывает что оно еще и работать будет быстрее так постарается вызвать все запросы пораньше(как работает async/await в циклах — не уверен, но что-то мне кажется что наш код станет псевдо-синхронным для этого цикла)

```

const Promise = require('bluebird')

function gogogo(queryList) {
  return Promise
    .resolve(queryList)
    .map(getType)
    // can be incapsulated in a function but original sample doesn't do that
    .map(function(type) {
      if (type === 1) {
        return getJson(query)
      }
      else if (type === 2) {
        return getRawData(query)
          .rawToJson(rawData)
      }
      else {
        return {}
      }
    })
    .reduce(Object.assign, {})
}

function doSomething() {
  gogogo(queryList)
    .then(templateGenerator)
    .tap(console.log)
}

doSomething()

```

 HaMI  11.04.17 в 17:51    

пока писал это понял — промисы нравятся тем кто предпочитает .map, .filter, .reduce циклам и наоборот async/await — любителям циклов

Хорошо, это или плохо — не знаю, это тема для извечного холивара. Но мне все же кажется, что async/await — позволяет писать в псевдо-синхронной манере игнорируя понимаю когда произойдет асинхронная операция

ну и заодно интересно — есть ли шанс достичь аналогичного поведения с async/await? с concurrency = Infinity или, например, concurrency = 1

 raveclassic  11.04.17 в 18:03    

Вот вам классическая задача.

Есть у вас массив урлов. Запрос по каждому из них возвращает ключ, и каждый, начиная со второго, принимает в запросе ключ, получен предыдущего запроса. Количество урлов не известно. Нужно получить последний ключ. И давайте договоримся, всевозможные хелперы всяких bluebird и компании использовать нельзя.



 raveclassic 11.04.17 в 18:07    

Ради интереса, представим, что у нас еще очень маленький стек, а урлов может быть больше 10к.

 HaMI  11.04.17 в 18:08    

давайте договоримся что вы будете использовать — Pascal или js из 7 осла? Неиспользовать технологии — это луддизм

Кроме того, начните сначала с себя. Ваш комментарий без кода выглядит, мягко говоря, голословно

 raveclassic  11.04.17 в 18:20    





Неиспользовать технологии – это луддизм

Мы как раз обсуждаем технологию: async/await или промисы. Не надо тащить сюда ворох библиотек, облегчающих работу с промис инкапсулирующих бойлерплейт.

Вот вам пример.

```
declare const get: (url: string, key: string) => Promise<string>;
const getKey = async (initial: string, urls: string[]): Promise<string> => {
  let result = initial;
  for (const url of urls) {
    result = await get(url, result);
  }
  return result;
};
```

PS. простите мне мой TS

 **NaMI** 11.04.17 в 18:51    

>Не надо тащить сюда ворох библиотек
одну и странно слышать это от человека предоставившего пример на typescript

```
const Promise = require('bluebird')






const getKey = (initial, urls) =>
  Promise.resolve(urls)
    .reduce(
      key, url => get(url, key),
      initial
    )
```

 **raveclassic** 11.04.17 в 18:56    

Пожалуйста, вот вам обычный ES:

```
const getKey = async (initial, urls) => {
  let result = initial;
  for (const url of urls) {
    result = await get(url, result);
  }
  return result;
};
```

Что вы везде этот bluebird тащите? Можете руками написать асинхронный редьюс?

 **NaMI** 11.04.17 в 19:01    

>Что вы везде этот bluebird тащите?
очевидно – нормальная обработка ошибок и приятные фичи

>Можете руками написать асинхронный редьюс?
думаю смогу, но давайте не будем это проверять. Потому что я попрошу взамен гарантий что вы сможете руками добавить язык async/await

 **raveclassic** 11.04.17 в 19:03    


Кроме того, начните сначала с себя.

думаю смогу, но давайте не будем это проверять.

Вы что-то мечетесь, неужели так сложно без bluebird?

Потому что я попрошу взамен гарантий что вы сможете руками добавить в язык async/await

Они уже в языке и в спеке.

 **mayorovp** 11.04.17 в 19:08    

reduce тоже есть в языке и в спеке.

 **raveclassic** 11.04.17 в 19:12    



Асинхронный?

 **mayorovp**  11.04.17 в 19:16   

Если вам позарез нужен асинхронный — держите:

```
function reduceAsync(array, fn, initial) {
  return array.reduce(
    (prev, current, index) => prev.then(prev2 => fn(prev2, current, index, array)),
    Promise.resolve(initial)
  );
}
```

Но я всегда простым обходился.

 **HaMI** 11.04.17 в 19:24    

Так что по рукам или слились? вы берете рантайм от 51-го Firefox и допиливаете туда async/await

А если серьезно, давайте вы мне не будете рассказывать — что я должен, а что нет.

>неужели так сложно без bluebird?

aga, а еще без underscore, jquery и еще вагона библиотек, которые в той или иной мере вошли в язык, Browser API


На этом — давайте закончим дискуссию, а то она несколько в другую плоскость перерешла.

 **raveclassic** 11.04.17 в 19:31    

Так что по рукам или слились? вы берете рантайм от 51-го Firefox и допиливаете туда async/await

Это вы слились. Вы тащите стороннее апи для расширения существующей спецификации только для того, чтобы упорно пытаться решить задачу, для которой в язык введен отдельный инструмент, не используя этот инструмент. Главное преимущество async/await и генераторов — возможность удобно работать с циклами. Вы так здорово приводите примеры на промисах в ситуациях, когда с их помощью действительно можно решить поставленную задачу, примера с циклом я от вас так и не дождался.

Ну что ж, действительно, закончим.

 **TheShock** 11.04.17 в 20:07    

очевидно — нормальная обработка ошибок и приятные фичи

То есть, встроенные в язык Промисы по вашему — не юзабельны?

 **unel**  11.04.17 в 19:00    

ну вот без await

```
const getKey = (prevKey, remainingUrls) => {
  if (!remainingUrls.length) {
    return Promise.resolve(prevKey);
  } else {
    return getKeyFromApi(remainingUrls[0]).then(newKey => {
      return getKey(newKey, remainingUrls.slice(1))
    });
  }
}
```

 **raveclassic** 11.04.17 в 19:04    

Я выше написал про маленький стек. Можете без рекурсии?

 **unel** 11.04.17 в 19:20    

насколько я помню, в случае асинхронных операций, стек не съедается таким образом (я даже проверил это для 18000 url текущей глубины стека в 17800)...

где-то даже видел "хак" в виде tail-call optimisation, когда себя же вызывают через setTimeout)

вот по памяти, я думаю, будет не очень оптимально, это да.



mayorovp 11.04.17 в 19:27



Чтобы было оптимально по памяти — надо вместо slice использовать растущий индекс. Получится даже оптимальнее чем через reduce.



unel 11.04.17 в 19:31



там кроме слайсов создаётся ещё куча замыканий в виде анонимок, вызывающих эту функцию с новыми параметрами



mayorovp 11.04.17 в 20:53



Да, но в каждый момент времени в памяти находится всего одно замыкание. Это очень небольшая нагрузка на память в отличие от пересозданий массива.



raveclassic 11.04.17 в 19:35



Я вам тут отвечу.

Спецификация промисов явным образом требует от рантайма чтобы колбек, переданный в then, запускался на пустом стеке (формально — не содержащем никаких пользовательских фреймов).

Про спешку и пустой стек не знал, спасибо. Я не особо силен в этой внутренке, но разве аргументы функции не образуют новый скоуп, который где-то должен лежать, даже при очистке стека? Это уже к вопросу расхода памяти.



raveclassic 11.04.17 в 19:39



Так же нужно действительно хранить замыкания, как написали выше.



mayorovp 11.04.17 в 20:55



Цепочка хранимых скоупов не может быть длиннее чем вложенность кода в редакторе и не зависит от глубины вызовов

И таких цепочек хранится всего одна — для следующей итерации.



raveclassic 11.04.17 в 21:37



Цепочка хранимых скоупов не может быть длиннее чем вложенность кода в редакторе и не зависит от глубины вызовов

Стоп, как это не зависит от глубины? Вам в рекурсивную функцию в каждом вызове приходят новые значения, которые должны храниться.



mayorovp 13.04.17 в 17:29



Но это каждый раз будет новая цепочка, с той же самой глубиной.



raveclassic 13.04.17 в 17:35



Ну так дело в то не в глубине, а в их количестве. На каждый виток рекурсии мы выделяем память заново под новые аргументы. И ради чего, чтобы не использовать await?



mayorovp 13.04.17 в 17:44



Вы так говорите как будто await ничего не выделяет при вызове :-)



raveclassic 13.04.17 в 18:05



Выделяет, но цикл — не рекурсия.



mayorovp 13.04.17 в 18:09



Опять двадцать пять... Почему вы считаете рекурсию безусловно хуже цикла, если она не требует больше памяти и не ест стек?



raveclassic 13.04.17 в 18:52



Ну где она не есть память-то?

Но это каждый раз будет новая цепочка,

Старая куда денется? Уничтожится? Зачем наматывать эти цепочки рекурсивно, когда можно просто пройти по массиву циклом, сохраняя ссылку на ключ?

Сравните 2 примера выше — в рекурсивном варианте вы выделяете память каждый раз заново для всей на getKey при вызове. Слайс массива — это вообще надругательство.



mayorovp 13.04.17 в 19:26



Про слайс массива я уже тоже писал, его надо на переменную с индексом заменить или на использование итератора.

Прошу обратить внимание, что все существующие js-реализации async/await используют рекурсивный вызов каждый оператор await. И нет оснований предполагать что рантайм браузера будет реализован сильно оптимальнее.



raveclassic 13.04.17 в 19:39



Await основан на генераторах, которые реализованы нативно в самом движке. Так что там с памятью все в порядке (надеюсь).



HaMI 11.04.17 в 19:04



оно рекурсивное – вам сейчас расскажут про то что ulгов больше чем размер стека. Вы можете это пофиксить извернувшись .catch – но будет выглядеть уродски



mayorovp 11.04.17 в 19:15



Нет, эта функция — не рекурсивная и прекрасно работает на ограниченном стеке.

Спецификация промизов явным образом требует от рантайма чтобы колбек, переданный в then, запускался на пустом стек (формально — не содержащем никаких пользовательских фреймов).



HaMI 11.04.17 в 19:33



вы правы, я как-то об этом не подумал.



Shannon 11.04.17 в 18:18



О том и речь, понадобился функционал сторонней библиотеки

А вот это я с трудом представляю как изобразить, чтобы такой вариант заработал:

```
return getRawData(query)
  .rawToJson(rawData)
```

При то, что и getRawData и rawToJson асинхронные функции. Либо тут надо быть гуру промисов, чтобы в этом разобраться

Но даже при этом всё равно получилось не то же самое, так как «return {}» не тоже самое что «break», и в случае с map перебор продолжит мой случае предполагался выход из цикла



raveclassic 11.04.17 в 18:21



Либо тут надо быть гуру промисов, чтобы в этом разобраться

Нет, это просто опечатка, должно быть `getRawData(query).then(rawToJson)`



Shannon 11.04.17 в 18:28



Тогда понятно, да



HaMI 11.04.17 в 18:40



да, очепятка – такое бывает с кодом который нельзя протестировать

```
return getRawData(query)
  .then(rawToJson)
```

>Но даже при этом всё равно получилось не то же самое, так как «return {}» не тоже самое что «break», и в случае с map перебор продолжится, а в моей случае предполагался выход из цикла

как-то это не логично чуть-чуть(точно не continue)

но ок:

```
const Promise = require('bluebird')
```



```
function gogogo(queryList) {
  return Promise
    .resolve(queryList)
    .map(getType)
    .then((types) =>
      types.find(x => x !== 1 && x !== 2)
    )
    .map(function(type) { // can be incapsulated in a function but original sample doesn't do that
      if (type === 1) {
        return getJson(query)
      }
      else {
        return getRawData(query)
          .rawToJson(rawData)
      }
    })
    .reduce(Object.assign, {})
}

function doSomething() {
  gogogo(queryList)
    .then(templateGenerator)
    .tap(console.log)
}

doSomething()
```

>функционал сторонней библиотеки

мы, вроде бы, обсуждаем фичу которая почти везде возможна только с транспILERом. Давайте, если на то пошло откажемся от всех библиотек сразу тогда. Представьте как будет интересно – все руками. Заодно и денег будем больше рубить с любого проекта.

Будет как 10 лет назад: «Ищу php-разработчика со своим фреймворком – любителей писать с нуля, просим не беспокоить»

 raveclassic 11.04.17 в 18:50    






Давайте, если на то пошло откажемся от всех библиотек сразу тогда. Представьте как будет интересно – все руками.

И смешались в кучу кони люди. Не путайте часть спецификации языка и сторонние библиотеки.

Async/await не требует никаких библиотек для работы. Генераторы *почти* не требуют никаких библиотек для работы. *Почти*, потому-ч можно раннер либо самому написать, либо взять из бабеля (в комментах выше), либо из со.

мы, вроде бы, обсуждаем фичу которая почти везде возможна только с транспILERом.

Раз, два, три.

 HaMI 11.04.17 в 18:56    

Мы же помним что это js? – тут все сначала стороная библиотека, а потом идет в язык

 Shannon 11.04.17 в 18:52    






мы, вроде бы, обсуждаем фичу которая почти везде возможна только с транспILERом

Нее, мы обсуждаем фичу которая нативно работает в node, и во всех актуальных браузерах, кроме Edge и Opera Mini
<http://caniuse.com/#feat=async-functions>

как-то это не логично чуть-чуть(точно не continue)
но ок:

Ну вот, про это и речь, «Уже становится немного сложновато представить на промисах, особенно если какой-то еще логики надо добаи

Всё становится очень хрупким и нужно всю цепочку держать в голове, чтобы небольшие изменения внести. При этом никто не говорит это не возможно, возможно-то возможно

 HaMI 11.04.17 в 19:46    

я вам привел как это будет выглядеть, сложности тут особой нет.

Дальше – рассуждайте сами, какой подход вам более покажется приемлемым

вопрос исключительно в том что стоит показывать равносильные примеры, а не те в которых один подход явно выгодней другого, н не к вам вопрос – а к статье



Shannon 11.04.17 в 20:57



Я про то и говорю:

Всё становится очень хрупким и нужно всю цепочку держать в голове, чтобы небольшие изменения внести. При этом никто не говорит, что это не возможно, возможно-то возможно

Достаточно чуть усложнить условие (приложение в процессе динамического развития, много экспериментов):

```
const type = await getType(query)
const subtype = await getSubType(query)
if (type === 1 && subtype === 4) {
  json = await getJson(query)
}
else if (type === 2 && !subtype) {
  const rawData = await getRawData(query)
  json = await rawToJson(rawData)
}
else if (subtype === 1) {
  break
}
```

И всё становится чуточку сложнее и нужно поломать голову, как это впихнуть в промисы без промисс-хелла.

И я уверен, что это получится сделать, но я не уверен, что можно это сделать за 10 секунд, как это делается на async-await

Но забыть про промисы нельзя хотя бы потому, что с помощью промисов удобно оборачивать функции на коллбэках в асинхронн функции.



mayorovp 11.04.17 в 21:16



Вот зачем вы пишете `Promise.resolve(queryList).map(getType)` — когда можно написать `queryList.map(getType)`? И будет раб без сторонних библиотек.

Вы же сначала сами переусложняете код — а потом жалуетесь что вас не понимают.

PS дихотомия между отверткой и молотком — ложная. Нужны оба инструмента!

```
function gogogo(queryList) {
  const items = queryList.map(query => {
    const type = await getType(query);

    if (type === 1) {
      return getJson(query)
    }
    else if (type === 2) {
      const rawData = await getRawData(query)
      return rawToJson(rawData)
    }
  })

  let resultJson = {}
  for (const item of await Promise.all(items)) {
    if (item === undefined) break;

    Object.assign(resultJson, item)
  }
  return resultJson;
}
```

И да, я знаю что этот код не полностью эквивалентен исходному.



Shannon 11.04.17 в 21:18



Вот этот вариант мне нравится



raveclassic 11.04.17 в 21:33



Вот оно! Есть правда некоторые неточности с отсутствующими кое-где `async` и `await`, но посыл-то верный, каждый инструмент ввел язык для своих целей и использовать нужно все.



mayorovp 11.04.17 в 21:44



Нет, `await` у меня все на месте! А вот про `async` я и правда позабыл напрочь...



raveclassic 11.04.17 в 21:51



getJSON же асинхронный? Мм, ну по логике, из названия. Хотя по изначальному коду и не скажешь.

Edit: Ааа, items это же промисы :)



HaMI 11.04.17 в 23:12



>Вот зачем вы пишете `Promise.resolve(queryList).map(getType)` — когда можно написать `queryList.map(getType)`?

Эти два куска кода совсем не эквиваленты. Посыл был показать `concurrent` и что выполнить `n`-запросов один за другим — это далеко тоже самое что выполнить `n`-запросов «сразу». И что это важно — понимать эту разницу. Но это никто в упор не хочет видеть.

проблема вашего и моего кода в другом — мы делаем лишние вызовы к `getType`.

>PS дихотомия между отверткой и молотком — ложная. Нужны оба инструмента!

Отличная мысль! давайте еще раз глянем в заглавие статьи. А потом посмотрим на примеры из нее — я не вижу там того что показ@
@Shannon, но не смог сформулировать

А нужно было показать что в случаях с циклами с пост-условием(вот этот `break`) нам, таки, намного легче писать с `async/await` — и п что это некрасиво, не-functional и вообще почти GOTO. И, наверное есть еще подобные примеры

В статье и комментариях я вижу как раз то о чем вы написали:

```
Promise.resolve([1, 2, 3]).map(getType)
```

сложнее чем(и впридачу, эквивалентно)

```
[1, 2, 3].map(query => {
  const type = await getType(query);
  return type
})
```

только в действительности первый вариант будет выполняться ~`n` времени(конечно, с оговорками), а второй `3*n` времени. И если эту разницу игнорировать — то нам вообще не нужен `async/await`(как и `promise`), нам просто нужно синхронное IO

Если я что-то не понимаю — прошу поправить



raveclassic 11.04.17 в 23:57



Вы придираетесь к разнице последовательного и параллельного выполнения. Да, разница видна, конечно же. Оба варианта мож
сделать на промисах, просто дело в том, что последовательные циклы на промисах пишутся в разы сложнее, чем на `async/await`
как последний прекрасно позволяет это сделать в читабельном и поддерживаемом формате.

И дело не в функциональщине vs императивщине, а в том, что оба подхода лучше сочетать, а не впадать в крайности.



Shannon 12.04.17 в 00:32



Все эти примеры выдуманы, чтобы показать упрощение синтаксиса, но вы зацепились за конкурентность

По сути, обычно следующий запрос может зависит от ответа предыдущего, а иногда и сразу от двух предыдущих запросов, поэто
конкурентность не требуется. А когда она требуется, то просто делается все через `Promise.all`

Вот упрощенный пример из рабочего проекта:

```
const [userdata, params] = await Promise.all([getUserData(), getParams()])
let bricks = []

if (params.allowedExtraData && userdata.needExtraData) {
  userdata.extraData = await getExtraData(userdata.id)
}

bricks.push(...params.bricksForModule[userdata.moduleType])
bricks.push(...params.bricksForType[params.mainType])

if (params.groups[userdata.groupType].isModerator) {
  bricks.push(...params.templatesModerator)
}
```

```
const bricksData = await Promise.all(bricks)
...
```

И дальнейшая обработка результатов

 **HaMI** 12.04.17 в 01:11 # 1 2 3 4

у вас кажись бажина в коде проекта:

```
bricks.push(...patams.templatesModerator)
```

 **Shannon** 12.04.17 в 02:04 # 1 2 3 4

Фраза «упрощенный пример» о чем нибудь говорит?

 **HaMI** 12.04.17 в 02:53 # 1 2 3 4

нет – со стороны выглядит будто вы этот кусок вырезали из кода, не меняя имен переменных. Но, в принципе, не суть. Не хотел вас поддеть или как-то обидеть

 **Shannon** 12.04.17 в 02:54 # 1 2 3 4

Ну тоесть вы серьезно думаете, что кусок кода с необъявленной переменной может скомпилироваться?

 **HaMI** 12.04.17 в 03:01 # 1 2 3 4


я ничего не думаю – увидел опечатку, сказал вам об этом. Вам достаточно было проигнорировать это или просто сказать – опечатка.

В следующий раз буду ставить много много смайликов – чтобы вы не подумали что я хочу вас обидеть

 **TheShock** 12.04.17 в 03:19 # 1 2 3 4

кусок кода с необъявленной переменной может скомпилироваться?

Это ведь JavaScript, конечно может. Или не JavaScript?

 **HaMI** 12.04.17 в 03:29 # 1 2 3 4

ключевое слово здень – необъявленная. Тут можно выдумать каким образом это прокатило – но будет выгляде натянута, и скорее как оправдание.

 **TheShock** 12.04.17 в 04:31 # 1 2 3 4

Не надо ничего выдумывать, оно просто будет корректно работать, это же JavaScript

 **Shannon** 12.04.17 в 03:46 # 1 2 3 4

Это ведь JavaScript, конечно может. Или не JavaScript?

Так и есть, мой промах, не учел что этот кусок кода в условии, которое может никогда не выполниться

 **bano-notit** 12.04.17 в 13:39 # 1 2 3 4

Прикол в том, что даже если условие будет положительным вместо этой переменной подствиться undefined. дальше уже соответствующие ошибки полезут, аля undefined is not a function, cannot read property <...> of undefined.

 **Shannon** 12.04.17 в 16:55 # 1 2 3 4


Да нее, будет «ReferenceError: patams is not defined» в любом случае, даже если без 'use strict' запускать

Единственное что, если без 'use strict' запускать, можно записать какое-нибудь значение в необъявленную переменную, она в глобальную область видимости попадет, но само по себе undefined не подставится

 **Shannon** 12.04.17 в 03:45 # 1 2 3 4

нет – со стороны выглядит будто вы этот кусок вырезали из кода

Как раз наоборот, код сократил раза в 3, а все переменные переименовал, чтобы легче смысл передать

 **NaMI** 12.04.17 в 02:40 # [иконка] [иконка] [иконка] [иконка]



Я не понял послыл вашего комментария.

с точкой зрения:

>PS дихотомия между отверткой и молотком — ложная. Нужны оба инструмента!

я вполне согласен — ваш первый пример, если его вдумчиво «покурить», показал что есть ситуация в которой `async/await` упрощает жизнь. Но в статье об этом ни слова.

Про ваш второй пример — я такого сказать не могу. Ну заменили вы `.then` на `await`. Вроде как стало на один уровень вложенности меньше, но стоит вам добавить обработку исключений и все вернуться на свои места.

то что вам не нужно пробрасывать через `.then` пары аргументов — это плюс.

Ну и он странно написан — `await Promise.all(bricks)`. возможно вы где-то кусок кода удалили — но, даже так, это странно складывать промисы и что-то «статическое» в один и тот же список. Скорее кто-то на всякий случай обернул чтобы не думать что там внутри. Есть еще вариант, что там в `params` промисы где-то лежат — но это тоже странно.

>По сути, обычно...

бывает по разному. давайте все же смотреть на картину в целом.

>А когда она требуется, то просто делается все через `Promise.all`

Если вас не затруднит, проиллюстрируйте свою идею кодом — по возможности, перепишите код из этого комментария https://habrahabr.ru/company/ruvds/blog/326074/#comment_10166298, само собой — с учетом опечатки и без `break` так чтобы де-юре как можно больше запросов «параллельно»

 **Shannon** 12.04.17 в 03:28 # [иконка] [иконка] [иконка] [иконка]



И Остапа понесло...

Ну и он странно написан — `await Promise.all(bricks)`

`bricks` — это набор функции, которые возвращают промис.

Это модули, которые делают только свою маленькую работу, все они запускаются параллельно и их результат складывается в единый результат, а дальше в шаблонизаторе, уже в зависимости от наличия модуля будет активирован или нет нужный кусок шаблона

Ну заменили вы `.then` на `await`. Вроде как стало на один уровень вложенности меньше, но стоит вам добавить обработку исключений и все вернуться на свои места.

В данном случае, обработка исключений одна единственная на более высоком уровне (на точке входа), на каждом этапе она требуется

>А когда она требуется, то просто делается все через `Promise.all`

Если вас не затруднит, проиллюстрируйте свою идею кодом — по возможности, перепишите код из этого комментария

Вот этот пример: https://habrahabr.ru/company/ruvds/blog/326074/#comment_10167050

Но если вам нужен именно пример моего изначального примера:

[Заголовок спойлера](#)

 **NaMI** 12.04.17 в 04:11 # [иконка] [иконка] [иконка] [иконка]



>`bricks` — это набор функции, которые возвращают промис.

`params.bricksForModule[userdata.moduleType]` — то есть это `promise`? или функция — если функция, то когда она вызывает

За пример кода — спасибо. В принципе увидел, то что хотел.

 **Shannon** 12.04.17 в 04:24 # [иконка] [иконка] [иконка] [иконка]



Да, это массив промисов, через `rest` параметр они вливаются в общий массив промисов, которые потом вызываются параллельно

 **NaMI** 12.04.17 в 04:35 # [иконка] [иконка] [иконка] [иконка]



Типа комбинированный подход.

из плюсов:

не меняет порядок элементов(у вас `promiseList` «как бы упорядочен»)

```
const Promise = require('bluebird')

function gogogo(queryList) {
  const parts = []
```

```
for (const query of queryList) {
  const type = await getType(query)
  if (type === 1) {
    parts.push(getJson(query))
  }
  else if (type === 2) {
    parts.push(
      getRawData(query).rawToJson(rawData)
    )
  } else {
    break
  }
}
return Promise
  .all(parts)
  .reduce(Object.assign, {})
```

В общем – спасибо за пример. Ну и всегда приятно пообщаться с человеком который подкрепляет свои суждения кодом. Приятных



mayorovp 12.04.17 в 05:43



только в действительности первый вариант будет выполняться ~n времени(конечно, с оговорками), а второй 3*n времени. И если эту разницу игнорировать – то нам вообще не нужен async/await(как и promise), нам просто нужно синхронное IO

Я ни коим образом не предлагаю эту разницу игнорировать! Напротив, я предлагаю совершенно другое: *использовать async/await когда нужно последовательное выполнение и then + Promise.all когда нужно параллельное.*

Только полноправные пользователи могут оставлять комментарии. Войдите, пожалуйста.

САМОЕ ЧИТАЕМОЕ

Сутки

Неделя

Месяц

Самое сложное в программировании это...

↑ +68 👁 32k 📖 240 💬 81

Необразованная молодёжь. Ответ бизнеса

↑ +99 👁 32,8k 📖 97 💬 801

Как мы торговали играми из киосков с газетами

↑ +95 👁 14,2k 📖 36 💬 85

Скажи «нет» Electron! Пишем быстрое десктопное приложение на JavaFX

↑ +44 👁 9,8k 📖 82 💬 80

Так ли легко переехать в Германию? Моя личная статистика поиска работы

↑ +24 👁 15,5k 📖 55 💬 94

ИНТЕРЕСНЫЕ ПУБЛИКАЦИИ

Необразованная молодёжь: попытка подытожить и немного личного

↑ +9 👁 2,1k 📖 14 💬 13

Как работает буфер обмена в Windows

↑ +6 👁 2k 📖 30 💬 3

Монады для Go-программистов

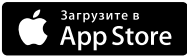


↑ +14 👁 2k 📖 27 💬 6

НурерX на Игромире-2017: часть 1 — SSD и DRAM GT

↑ +14 👁 1k 📖 1 💬 1

Семантическая разметка: LaTeX, DocBook или ???

↑ +8 👁 1,1k 📖 10 💬 8

Аккаунт	Разделы	Информация	Услуги	Приложения
Войти	Публикации	О сайте	Реклама	<div> </div>
Регистрация	Хабы	Правила	Тарифы	
	Компании	Помощь	Контент	
	Пользователи	Соглашение	Семинары	
	Песочница	Конфиденциальность		
 © 2006 – 2017 «TM»		Служба поддержки	Мобильная версия	