



39,05

Рейтинг

ZeroTech

Разрабатываем сложные web-проекты.



RA_ZeroTech 9 декабря 2016 в 19:59 Разработка

Путеводитель по JavaScript Promise для новичков

Разработка веб-сайтов, JavaScript, Ajax, Блог компании ZeroTech

Tutorial



Этот материал мы подготовили для JavaScript-программистов, которые только начинают разбираться с «Promise». **Обещания (promises)** в JavaScript – это новый инструмент для работы с отложенными или асинхронными вычислениями, добавленный в ECMAScript 2015 (6-я версия ECMA-262).

До появления «обещаний» асинхронные задачи можно было решать с помощью функций обратного вызова или с помощью обработки событий. Универсальный подход к решению асинхронных задач – обработка событий. Менее удобный, но также имеющий право на существование, использовать функции обратного вызова. Конечно, выбор решения зависит от стоящей перед вами задачи. Вариант решения задач с помощью «обещаний», скорее, призван заменить подход к функциям обратного вызова.

В использовании функций обратного вызова есть существенный недостаток с точки зрения организации кода: **"callback hell"**. Этот недостаток заключается в том, что в функции обратного вызова есть параметр, который, в свою очередь, также является функцией обратного вызова – может продолжаться до бесконечности.

```

1  /**
2   * пример callback hell
3   * @param err
4   * @param callback
5   */
6  var asyncAgentSmith = function (err, AgentSmith)
7  {
8      AgentSmith(function (err, AgentSmith2)
9      {
10         AgentSmith2(function (err, AgentSmith3)
11         {
12             AgentSmith3(function (err, AgentSmith4)
13             {
14                 AgentSmith4(function (err, AgentSmith5)
15                 {
16                     AgentSmith5(err, function ()
17                     {
18                         // ...
19                     });
20                 });
21             });
22         });
23     });
24 };
25
26
27
28
29
30
31
32

```

Может образоваться несколько уровней таких вложенностей. Это приводит к плохому чтению кода и запутанности между вызовами функции обратного вызова. Это, в свою очередь, приведет к ошибкам. С такой структурой кода найти ошибки очень сложно.

Если все же использовать такой подход, то более эффективно будет инициализировать функции обратного вызова отдельно, создавая и в нужном месте.

Давайте рассмотрим работу «обещаний» на примере конкретной задачи:

После загрузки страницы браузера необходимо показать изображения из указанного списка.

Список представляет собой массив, в котором указан путь к изображению. Например, для показа изображений в слайдере вашей баннерной системы на сайте или асинхронной загрузки изображений в фотоальбоме.

```

/**
 * список изображений
 * (предположим, что изображения 1.jpg, 2.jpg, 3.jpg, 4.jpg существуют, а
 * fake.jpg - нет)
 *
 * @type {string[]}
 */
var imgList = ["img/1.jpg", "img/2.jpg", "img/fake.jpg", "img/3.jpg", "img/4.jpg"];

```

Сначала напишем функцию, которая подгружает одно изображение по указанному url.

function loadImage(url)

```

/**
 *
 * подгружаем изображение по указанному url
 *
 * @param url
 * @returns {Promise}
 */
function loadImage(url)
{
    //объект "обещание"
    return new Promise(function(resolve, reject)
    {
        var img = new Image();
    });
}

```

```
img.onload = function()
{
    //в случае успешной загрузки изображения, результат "обещания" будет url этого изображения
    return resolve(url);
}
img.onerror = function()
{
    //в случае не успешной загрузки изображения, результат "обещания" будет url этого изображения
    return reject(url);
}
img.src = url;
});
}
```

Объект «обещание» создается с помощью конструктора `new Promise(...)`, которому в качестве аргумента передается анонимная функция с параметрами: `resolve`, `reject`. Они, в свою очередь, так же являются функциями. `Resolve()` — сообщает о том, что код выполнен «успешно», — код выполнен с «ошибкой» (что считать «ошибкой» при выполнении вашего кода, решать вам. Это что-то вроде `if(true){...} else {...}`).

Интерфейс `Promise` (обещание) представляет собой обертку для значения, неизвестного на момент создания обещания. Он позволяет обрабатывать результаты асинхронных операций так, как если бы они были синхронными: вместо конечного результата асинхронного метода возвращается обещание, результат которого можно получить в некоторый момент в будущем.

При создании обещание находится в ожидании (состояние `pending`), а затем может стать выполнено (`fulfilled`), вернув полученный результат (значение), или отклонено (`rejected`), вернув причину отказа.

В методы `resolve()` и `reject()` можно передавать любые объекты. В метод `reject()`, как правило, передают объект типа `Error` с указанием причины ошибки («отклоненного» состояния «обещания»). В любом случае, это не обязательно. Решение, как дальше вы будете обрабатывать такие ситуации — за вами.

На данный момент может показаться, что «обещание» совершенно не нужно использовать в этой ситуации. Пока мы лишь устанавливаем индикатор того, было ли загружено изображение. Однако вскоре вы увидите, что этот механизм может легко, интуитивно понятно определить что произойдет после того, как задача будет выполнена (изображение загружено или нет).

Методы `then()` и `catch()`

Всякий раз, когда вы создаете объект «обещание», становятся доступны два метода: **`then()`** и **`catch()`**. Используя их, вы можете выполнить код при успешном разрешении «обещания» (`resolve(...)`) или же код, обрабатывающий ситуацию с «ошибкой» (`reject(...)`).

`then()` и `catch()`

```
function myPromise()
{
    return new Promise(function(resolve, reject)
    {
        //псевдо асинхронный код
        var async = true; //или false
        if (!async)
            return reject(new Error("не удалось выполнить..."));

        return resolve(1);
    });
}

myPromise()
    .then(function(res)
    {
        console.log(res); //выведет 1
    })
    .catch(function(err) {
        console.log(err.message); //выведет сообщение "не удалось выполнить..."
    });
```

Примечание: не обязательно возвращать (`return`) `resolve(...)` или `reject(...)`. В примере выше можно было бы написать так:

```
//псевдо асинхронный код
var async = true; //или false
if (!async)
{
    reject(new Error("не удалось выполнить..."));
}
else
{
    resolve(1);
}
```

В результате вызова `myPromise()` все равно сработал бы метод `then()` или `catch()`. Лучше всего завести сразу привычку — всегда возвращать `resolve(...)` или `reject(...)`. В будущем это поможет избежать ситуации, когда код будет работать не так, как ожидается.

В методы `then()` и `catch()` передают две анонимные функции. Синтаксис метода `then()` в общем случае такой:

```
then(function onSuccess(){}, function onFail(){});
```

Параметр `function onSuccess(){}` будет вызван в случае успешного выполнения «обещания», `function onFail(){}` — в случае ошибки. По этой причине следующий код будет работать одинаково:

Примеры метода then()

```
myPromise()
.then(function(res)
{
    console.log(res); //выведет 1
})
.catch(function(err){
    console.log(err.message); //выведет сообщение "не удалось выполнить..."
});

myPromise()
.then(function(res)
{
    console.log(res); //выведет 1
},
function(error)
{
    console.log(err.message); //выведет сообщение "не удалось выполнить..."
});

myPromise()
.then(function(res)
{
    console.log(res); //выведет 1
})
.then(undefined, function(error)
{
    console.log(err.message); //выведет сообщение "не удалось выполнить..."
});
```

Гораздо привычнее и понятнее использовать `catch(...)`. Также метод `catch()` можно вызывать «посередине» цепочки вызовов `then()`, если логика вашего кода того требует: `then().catch().then()`. Не забывайте вызывать `catch()` последним в цепочке: это позволит вам всегда отлавливать «ошибочные» ситуации.

Вызовем наш метод `loadImage(url)` и для примера добавим одну картинку на страницу:

```
//считаем, что на странице есть элемент с id="images", например, div
loadImage(imgList[0])
.then(function(url)
```

```
{
    $('#images').append('');
})
.catch(function(url)
{
    //как и сообщалось выше, не обязательно, чтобы сюда передавался объект типа Error
    //например, вы захотите сохранить в отдельный массив пути к картинкам, которые не подгрузились, и потом что-нибудь сделать...
    console.log("не удалось загрузить изображение по указанному пути: ", url);
});
```

Последовательная рекурсивная подгрузка и отображение изображений

Напишем функцию для последовательного отображения изображений:

[function displayImages\(images\)](#)

```
/**
 * последовательная рекурсивная подгрузка и показ изображений
 *
 * @param images - массив с url
 */
function displayImages(images)
{
    var imgSrc = images.shift(); // проходим по массиву с изображениями
    if (!imgSrc) return; //если в результате рекурсии прошли по всему массиву

    //если в массиве еще есть изображение, загружаем его
    return loadImage(imgSrc)
    .then(function(url)
    {
        $('#images').append('');
        return displayImages(images); //рекурсия
    })
    .catch(function(url)
    {
        //если какое-то из изображений не загрузилось, переходим к следующему изображению
        console.log('не удалось загрузить изображение по указанному пути: ', url);
        return displayImages(images); //рекурсия
    });
}
```

Функция `displayImages(images)` последовательно проходит по массиву с url изображений. В случае успешной подгрузки мы добавляем изображение на страницу и переходим к следующему url в списке. В противоположном случае – просто переходим к следующему url в спис

Возможно, такое поведение отображения изображений не совсем то, что необходимо в данном случае. Если требуется показать все изобра только после того, как они были загружены, нужно реализовать работу с массивом «обещаний».

```
var promiseImgs = [];
promiseImgs = imgList.map(loadImage);

//для наглядности
promiseImgs = imgList.map(function(url) {
    return loadImage(url);
});
```

В массиве `promiseImgs` теперь находятся «обещания», у которых состояние может быть как «разрешено» так и «отклонено», так как изображение `fake.jpg` физически не существует.

Для завершения задачи можно было бы воспользоваться методом `Promise.all(...)`.

`Promise.all(iterable)` возвращает обещание, которое выполнится после выполнения всех обещаний в передаваемом итерируемом аргументе

Однако у нас в списке есть изображение, которого физически не существует. Поэтому методом `Promise.all` воспользоваться нельзя: нам необходимо проверять состояние объекта «обещание» (`resolved` | `rejected`).

Если в массиве «обещаний» есть хотя бы одно, которое «отклонено» (rejected), то метод Promise.all так же вернет «обещание» с таким состоянием, не дожидаясь прохождения по всему массиву.

Поэтому напомним функцию loadAndDisplayImages.

Подгружаем изображения, и показываем их на странице все сразу

[function loadAndDisplayImages](#)

```
/**
 *
 * @param imgList - массив url
 * @returns {Promise}
 */
function loadAndDisplayImages(imgList)
{
    var notLoaded = []; //сохраним url, какие не были загружены
    var loaded = []; //сохраним url, какие были загружены
    var promiseImgs = imgList.map(loadImage);

    //вернем результат работы вызова reduce(...) - объект Promise, чтобы можно было потом при необходимости продолжи
    //цепочку вызовов:
    //loadAndDisplayImages(...).then(...).catch(...);
    return promiseImgs.reduce(function (previousPromise, currentPromise)
    {
        return previousPromise
            .then(function ()
            {
                //выполняется этот участок кода, так как previousPromise - в состоянии resolved (= Promise
                resolve())

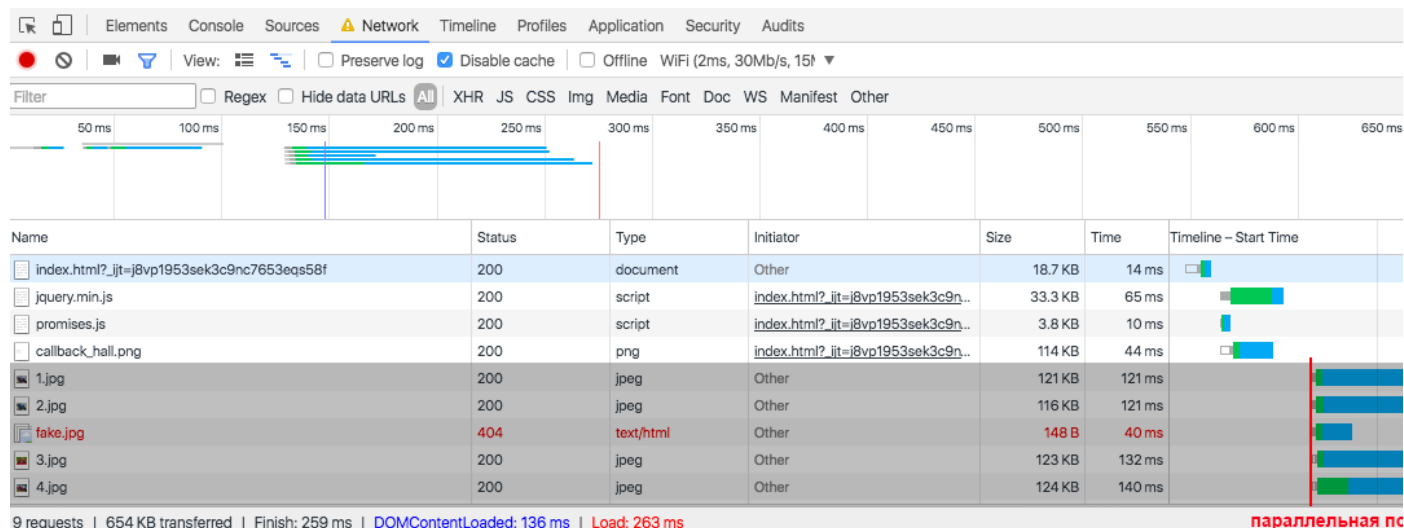
                return currentPromise;
            })
            .then(function (url) //для "обещаний" в состоянии resolved
            {
                $('#images').append('');
                loaded.push(url);
                return Promise.resolve(url);
            })
            .catch(function (url) //для "обещаний" в состоянии rejected
            {
                console.log('не удалось загрузить изображение по указанному пути: ', url);
                notLoaded.push(url);
                return Promise.resolve(url);
            });
    }, Promise.resolve())
        .then(function (lastUrl)
        {
            console.log('lastUrl:', lastUrl);

            let res = {loaded: loaded, notLoaded: notLoaded};

            //но мы вернем Promise, значение которого будет объект
            return Promise.resolve(res);
        });
}

loadAndDisplayImages(imgList)
    .then(function (loadRes)
    {
        console.log(loadRes);
    })
    .catch(function (err)
    {
        console.log(err);
    });
```

Можно посмотреть сетевую активность в браузере и убедиться в параллельной работе (для наглядности в Chrome была включена эмуляция подключения по Wi-Fi (2ms, 30Mb/s, 15M/s):



Разобравшись, как работать с Promise, вам будет проще понять принципы работы, например, с API Яндекс.Карт, или Service Worker – именно они используются.

UPD: В статье не озвучил один важный момент, с которым, отчасти, был связан совет писать `return resolve()` или `return reject()`.

Когда вызываются данные методы, «обещание» устанавливается в свое конечное состояние «выполнено» или «отклонено», соответственно. После этого состояние изменить нельзя. Примеры можно посмотреть в комментариях.

Метки: javascript, promise, разработка веб-сайтов, программирование

↑ +14 ↓ 291 47,5k 61

ZeroTech 39,05
 Разрабатываем сложные web-проекты.

5,0 **0,0** **1**
 Карма Рейтинг Подписчики

Алексей @RA_ZeroTech
Программист

Сайт Facebook

Поделиться публикацией

ПОХОЖИЕ ПУБЛИКАЦИИ

22 августа в 18:06

Мобильные браузеры и их пушистые лапки

↑ +17 9k 43 13

28 ноября 2016 в 18:36

11 видов кэширования для современного сайта

↑ +8 21k 193 17

ВАКАНСИИ КОМПАНИИ ZeroTech

Мой к






































































Full-stack разработчик PHP

Москва • Полный рабочий день


Вакансии компании

Создать ре


Комментарии 61


-  **svboobnov** 09.12.16 в 20:23   
- Спасибо за краткое, но довольно ёмкое введение.
Я с JavaScript сталкивался в 2004 году, потому мне полезно =)
-  **Fen1kz**  09.12.16 в 21:30   
- Интересно, а рассказы про промисы ещё актуальны? (Не сарказм, просто иногда подумываю, а не написать ли и мне статью, где я всё всем разжую них. Однако боюсь что все уже знают и закидают шапками типа "добро пожаловать в 2012")
-  **RA_ZeroTech** 09.12.16 в 21:44     
- А почему им не быть актуальными?) async await не так давно уж и появились, если на это намек в вопросе :) да и статья не об эволюции асинхрс программирования в целом, а только о ее части :)
-  **akzhan** 10.12.16 в 03:48     
- async/await при том не отменяют Promise, а скорее дополняют оные.
-  **Kain_Haart** 10.12.16 в 06:14     
- Промисы-то актуальны, вопрос в том, актуальны ли новые рассказы про промисы для новичков, и уж крайне спорно — называть промисы «новый инструмент» в декабре 2016
-  **Keyten** 10.12.16 в 16:49     
- Если честно, про промисы можно вообще всё рассказать буквально за минуту (или за абзац текста и кода), т.к. это просто-напросто иной синтаксис записи обработки событий. Так что меня удивляют большие статьи об этом. Сам научился работать с промисами, тупо проглядев кусок кода, в котором они использовались. Всё понятно и очевидно.
Имхо, конечно.
-  **RA_ZeroTech**  11.12.16 в 02:51     
- в чем-то Вы правы.
только кому-то достаточно прочитать документацию, кому-то ее может не хватить, возможно, из-за отсутствия тех примеров, которые будут понятны. статьи же пишут с целью поделиться опытом, который другим может помочь лучше разобраться в той или иной теме. иначе, можно всегда ссылаться на документацию, и не только по этой теме, а вообще.
-  **Singapura** 12.12.16 в 12:33     
- Вот хоть бы один из «учителей», Про-вёл-бы на примере «Hello» через все главы учебника JS меняя в динамике «цвета», «буквы», «шрифты», «циклы», «размеры»,... и под конец «выжал-бы» этот «Привет» через какой-нибудь порт по-битово... куда-нибудь...
-  **svboobnov** 12.12.16 в 23:03     
- Интересная мысль, надо попробовать. Как в том анекдоте: «Вы знаете, коллега, я третий год объясняю сопромат студентам, и даже с полностью понял!» =).
@Singapura, у Вас странное написание слов: принято писать не «по-битово» а «побитово», не «выжал-бы» а «выжал бы».
Если же Вы пытались поддеть @RA_ZeroTech, то зря: фразы «в чём-то» «кому-то» и «как-то так» «кем-то» пишутся именно так, с деф
-  **Singapura** 27.12.16 в 20:42     
- Привлекаю внимание))
-  **Singapura** 27.12.16 в 20:53     
- Взгляд студента, должен цепляться за знакомые конструкции в тексте... И фиксировать внимание на новых.
-  **MaxKorz** 12.12.16 в 12:33     

Только хабре статьей, подобной вашей (о Promises для новичков и/или профессионалов), уже около десятка. Очередная статья конечно будет актуальна, только вот зачем в очередной раз всё разговывать?

 **RidgeA** 09.12.16 в 21:38 # # # # #
В результате вызова `myPromise()` все равно сработал бы метод `then()` или `catch()`. Лучше всего завести сразу привычку — всегда возвращать `resolve` или `reject(...)`. В будущем это поможет избежать ситуации, когда код будет работать не так, как ожидается.


Можете пример привести?

 **Apathetic** 09.12.16 в 21:58 # # # # #
Присоединяюсь к вопросу. Никакой необходимости возвращать `resolve` или `reject` нет. Напротив, их весьма удобно использовать в качестве коллбэков при «промисификации» всяких коллбековых апи, и никаких `return` там, разумеется, и в помине нет.

 **RA_ZeroTech** 09.12.16 в 22:11 # # # # #
Начав писать ответ на вопрос, понял, что не акцентировал внимание на том, что после вызовов `resolve()` или `reject()` состояние «промиса» уже не изменить. то есть, вот этот код в обоих случаях выведет фразу вида «Promise rejected», несмотря на то, что в первом варианте нет «`return`» `resolve()` или `reject()`

В будущем это поможет избежать ситуации, когда код будет работать не так, как ожидается.

Этот фраза больше относилась к тем ситуациям, когда забывают возвращать «промис» в своих методах. При этом экспешина не возникает, напротив `return Promise`

 **RA_ZeroTech** 09.12.16 в 22:27 # # # # #
раз вызвали `resolve` или `reject()`, и «состояние промиса установлено», можно ошибочно предположить, что код, который следует после них, не будет выполнен. Но это не так. Пример: пример с `return resolve` или `reject()`

отчасти, и по этому тоже советовал всегда возвращать `return resolve()` или `reject()`, так как по логике, после вызовов этих методов, следующий за ним код не должен вызываться. иначе можно запутать самого себя, разбираясь в чем же дело...

PS. да, есть исключение для этого совета: если эти методы вызываются в качестве колбэков. как правильно заметил **Apathetic**

 **RidgeA** 10.12.16 в 00:24 # # # # #
`console.log('у нас же «ошибка», почему оказались здесь?');`


потому что промис — это конечный автомат, вызов `reject/resolve` меняет состояние, но не завершает выполнение текущей функции.

вот попробуйте:

```
new Promise((resolve, reject) => {
  setTimeout(() => {
    console.log('111111');
    resolve();
    setTimeout(() => {
      console.log('222222');
    });
  });
}).then(console.log.bind(null, '333333'));
```

Вот еще можно глянуть: <https://www.promisejs.org/implementing/>

 **RidgeA** 10.12.16 в 00:35 # # # # #
<http://caniuse.com/#search=Promise>

 **RA_ZeroTech** 10.12.16 в 00:48 # # # # #
потому что промис — это конечный автомат, вызов `reject/resolve` меняет состояние, но не завершает выполнение текущей функции.

совершенно верно :) именно это я и показал в примере



magicstream 10.12.16 в 00:27



а как дела обстоят с нативной поддержкой браузерами? уже все подтянулись?



novrm 10.12.16 в 01:09



При желании можете использовать, например bluebird.



Apathetic 12.12.16 в 22:36



Скорее даже не при желании, а при возможности — если позволяет «бюджет» (читай — запас по размеру кода), лучше использовать Bluebird, как он намного быстрее нативных промисов.



RA_ZeroTech 10.12.16 в 01:25



Chrome с версии 32, Firefox с 29-ой, Opera с 19-ой, Safari с 7.1, IE который Edge.

Поэтому стоит использовать полифилы...

@novrm рекомендует bluebird. Я, в свою очередь, тоже ей пользуюсь, в том числе и при написании серверного JS (NodeJS). Достаточно много пог методов реализовано.



jMas 10.12.16 в 11:53



А не лучше ли использовать какой ни будь полифил, который добавляем только объект window.Promise, и только те методы которые гарантируются присутствуют в браузерах. Ибо если вы завязываетесь на bluebird и на методы которых нет в реализациях браузеров — вы завязываетесь на кастомную реализацию промисов, а не полифилите реализацию браузера.



novrm 10.12.16 в 21:10



Promise, если не ошибаюсь — это технология...

Как она будет реализована — другое дело...

Но главное — должны быть полностью соблюдены рекомендации, что присутствует в bluebird, но отсутствует в том же jQuery...



jMas 10.12.16 в 21:50



Bluebird Promise да, поддерживает стандарт, но и выходят за его рамки. Если тебе нужен полифил, то тебе нужен полифил со стандартным набором методов. Потому что полифил нужен ровно до того момента, когда он становится не нужным. Тогда вы выкидываете полифил в пользу браузерной реализации. Если вы взяли в качестве "полифила" Bluebird и начали использовать методы не входящие в стандарт — выкинуть такой "полифил" не получится, и в этом случае полифил перестает быть полифилом.



novrm 11.12.16 в 00:42



Извините, но можно «локально» использовать Promise посредством bluebird...

Имею ввиду — использовать «стандартный» набор методов.

Если хотите, вот пример возможной реализации (es6).

```
/**
 * Import bluebird plugin.
 */
* @link https://github.com/petkaantonov/bluebird/
* @link http://bluebirdjs.com/docs/api-reference.html
*/
import BowerAssetBluebirdPlugin from 'asset/bluebird/js/browser/bluebird.min';

'use strict';

/**
 * Promise class wrapper.
 */
classLoader.autoload['BundleFramework/Promise/Promise'] = (function () {
  /**
   * Set private properties.
   */
  let _plugin = BowerAssetBluebirdPlugin;

  /**
   * Promise class.
   */
  return class {

    /**
     * Constructor. Create an instance.
     */
  }
})
```

```

* @param object config
*/
constructor(config = Object(config = {})) {
    Object.freeze(this);
};

/**
 * Ajax action.
 *
 * @link http://bluebirdjs.com/docs/coming-from-other-libraries.html#coming-from-jquery-deferreds
 * @param object options
 * @return object new _plugin
 */
ajax(options = Object(options = {})) {
    let plugin = this.getPlugin();

    return new plugin(function(resolve, reject) {
        $.ajax(options).done(resolve).fail(reject);
    });
};

getPlugin() {
    return _plugin;
};

};

})();
export {classLoader};

```

 jMas 11.12.16 в 02:01 # 📌 🔍 ↻

По-моему, достаточно подключить глобально файл-полифил и использовать `Promise` как часть глобального окружения (`global, window`). По-моему, это как раз и идея полифилов "доукомплектовать окружение отсутствующими в данной версии браузера компонентами". В вашем же случае, насколько я смог понять, используется какой-то кастомный прелоадер компонентов, через который вы и получаете доступ к `Promise`. В общем то, реализацию можно будет подменить и через него, но используя `Bluebird` вы (или ваши коллеги) рискуете завязаться на нестандартные методы.

И еще кейс, если вы будете использовать стороннюю библиотеку, которая не знает о вашем прелоадере компонентов — а в браузере внезапно в глобальном scope нет `Promise` — библиотека не заработает.

 novrm 11.12.16 в 14:49 # 📌 🔍 ↻

Не совсем понял ваши мысли об нестандартных методах?

Разработчики `Bluebird` как раз следуют стандарту...

Ну а «сахар» плагина — можете не использовать. Или явно его «выключить» — создав обертку над `Bluebird` (пример реализации которого я привел выше).

Кроме того — что такое стандарт? Это просто рекомендации...

Возможно «сахар» `Bluebird` завтра станет этим самым стандартом.

 jMas 11.12.16 в 15:43 # 📌 🔍 ↻

Или явно его «выключить» — создав обертку над `Bluebird` (пример реализации которого я привел выше).

Обертка плоха, потому что использует систему резолвинга зависимостей, наподобие `require.js`. Подключаешь стороннюю библиотеку, которая ожидает `window.Promise` и она не работает, например, в относительно стареньких браузерах. Поэтому и полифил который как раз создает недостающий объект окружения `window.Promise`.

Возможно «сахар» `Bluebird` завтра станет этим самым стандартом.

Стандарт принимает рабочая группа. И если там нет в планах добавлять методы из `Bluebird`, значит их не добавят.

Возможно «сахар» `Bluebird` завтра станет этим самым стандартом.

Например через пару лет мы захотим выкинуть поддержку браузеров где нет промисов, проводим рефакторинг и принимаем решение выкинуть полифил `Bluebird Promise`. Смотрим в код, и видим, что пару программистов заюзали по всему проекту его «сахар». Получится ли в этом случае выкинуть такой «полифил»?

 novrm 11.12.16 в 16:04 # 📌 🔍 ↻

Знаете, вы так мыслите, как будто ваш код будет работать без изменений 100 лет.

Через несколько лет относительно стареньких браузеров никто поддерживать не станет.

Более того, возможно через пару лет и самих промисов не станет...

Их заменят чем-то еще более универсальным... Например — генераторами.

Именно потому (ИМХО) резонно использовать «обертки» над технологиями...
Дабы отделить их использование от реализации.

А если вы желаете использовать некую стороннюю библиотеку, которая в свою очередь ожидает `window.Promise` — это прс это библиотеки, что она внутри не реализует полифил...
Всегда можно найти библиотеку, которая реализует полифил или самому ее допилить...



jMas 11.12.16 в 17:11

Именно потому (ИМХО) резонно использовать «обертки» над технологиями...

А если вы желаете использовать некую стороннюю библиотеку, которая в свою очередь ожидает `window.Promise` — это проблема это библиотеки, что она внутри не реализует полифил...

Она не обязана, если библиотека ожидает современное окружение. Ваша цель — предоставить соответствующее окруж или эмулировать его. В идеальном мире все окружение должно быть описано в зависимостях, но исторически такого механизма у нас нет. То есть, мы привыкли, что у нас есть `window`, `localStorage`, `location`. Конечно в идеале необходимо сделать декораторы для каждого, но часто это избыточно.

Знаете, вы так мыслите, как будто ваш код будет работать без изменений 100 лет.

Ну 100 лет преувеличено, но десятки лет — это почти реальность.



novrm 12.12.16 в 12:59

Для `Proху` и прочих новых технологий тоже полифил найдете?

А если не найдете?

Сознайтесь, что полифил не универсальный механизм внедрения новых технологий? Наверно нет.



jMas 12.12.16 в 16:12

Сознайтесь, что полифил не универсальный механизм внедрения новых технологий? Наверно нет.

Тут соглашусь.



alQlugin 11.12.16 в 09:24

На самом деле пока даже рекомендуется использовать `Bluebird` вместо нативных промисов. Если верить бенчмаркам, он в разы бы



jMas 11.12.16 в 10:33

Не спору, но если производительность `Promise` не сильно важна, я предпочитаю обстрагироваться и использовать нативные `Pr` или легкие полифилы. Потому как производительность нативной реализации могут существенно улучшить.

Просто в `Bluebird`, как я и писал, не нравится что API выходит за пределы стандарта, с одной стороны это хорошо, но как по мне, выглядит странно.



w4r_dr1v3r 10.12.16 в 01:09

Начинающим вроде меня тоже полезно прочесть побольше подобного материала. Всё неплохо структурировано и наглядно, на мой вкус. Автору бог спасибо!



slavugan 10.12.16 в 01:09

А как вы думаете, господа, не станет ли реактивный подход (`RxJS`) заменой промисов в будущем?



Rulexex 10.12.16 в 01:47

Эти ребята никак не могут определиться, как всё-таки правильно отменять промисы. Например, делаем одностраничное приложение, экран открыт какого-нибудь контента. Начали с промиса отправки запроса на сервер с запросом данных и закончили отрисовкой этого всего, накинув кучу `then'ов`, которые там внутри умудряются пять раз перепотрошить полученный контент, нарендерить двадцать шаблонов и чёрт знает что ещё.

Всё это дело понеслось, а тут пользователь нажимает на ссылку, старый контент уже никому не нужен, нужно загружать новый. Окей, создаём новы промис, повторяем накидывание обработчиков. Постойте-ка, у нас в процессе выполнения старый. И тут оказывается, что у **Promises/A+** нет никаки вариантов обработки отмены промиса.

Нам остаётся три пути:

- В каждом обработчике проверять, а не устарел ли запрос на действие, может результат уже никому не нужен.
- Сделать так, чтобы результат выполнения ВСЕГО промиса был проигнорирован (хотя и выполнен).
- Использовать 3rd party промисы или пилить свои.

Пока не особо думал об этом, на данный момент отошёл от промисов, давно не использовал, но пока выглядит так, что отмена должна найти стык, а промис ещё в процессе резолвинга, подменить ему then'ы на пустышки, и послать внутрь самого промиса сигнал отмены, который он может обработать чтобы прекратить своё выполнение, если может (к примеру, если там промис, который просто спит сколько-то времени, то в обработчике отмены он просто сделать `clearTimeout`).

Это всё ещё можно снабдить специальным сахаром для обработки отмен, чтобы можно было обрамлять некоторые куски цепочки операций, чтобы обрабатывать некоторые специфические случаи.

К примеру, есть метод, удаляющий некий `item`, возвращающий промис о завершении. Внутри он отправляет запрос на сервер, плюс делает какие-то действия с моделью. Пользователь жмёт «удалить», метод вызывается, промис начинает работу. Тут пользователь снова жмёт «удалить», очень резко настолько резко, что запрос даже не успел уйти на сервер (окей, может, у нас есть какая-нибудь логика группировки запросов, или окна общения с сервером, поэтому он не уходит сразу). Тогда мы должны отменить промис удаления. Но произведя отмену, нам бы хорошо знать, ушёл запрос на сервер или нет. Поэтому мы можем вставить в место, где совершается непосредственно запрос, специальный обработчик, например, `.onCancel`, который вызовется только когда чейн промисов в состоянии отмены и в него передастся, какое состояние у промиса, на который он непосредственно был на (т.е. начал он резолвиться или ещё нет). Где каким-то образом куда-нибудь сообщим, что да как. И, например, если запрос уже ушёл, то нам нужно послать второй в догонку, мол, «сервер, пользователь передумал это удалять, верни как было, пожалуйста».

Я пока не могу придумать, как можно типизированно сделать, чтобы тот, кто инициировал отмену мог подоставать данные из этих обработчиков с глубиной, ибо структура чейна промисов может быть совсем любая, и мы не всегда сможем сказать, какие данные наши, какие не наши (например, е чейне делается несколько запросов, у всех из которых есть обработчики отмены, генерирующие данные, мы отменяем чейн и хотим в результате операции отмены знать результат отмены конкретного типа запроса). Пока думается, что можно при инициации отмены передавать таблицу `Symbol`→обработчик, а потом очень аккуратно и обдуманно в обработчиках отмен посылать туда данные, если в сигнале отмены в таблице присутствует некоторый символ.

Ох, что-то случайно прорвало.

 Rulexec 10.12.16 в 01:54 # 📌 📄 🔄

> Тут пользователь снова жмёт «удалить», очень резко, настолько резко

Опечатка, жмёт «отменить удаление».

 Shifty_Fox 10.12.16 в 02:48 # 📌 📄 🔄

Эм.

У вас конкретная специфическая задача — прерывать длинную синхронную функцию (которая внутри асинхронная вся такая, но выглядит синхронно в этой функции нет цикла — она просто очень длинная).

Такая же проблема у вас бы была, если бы у вас была самая обычная функция на много строк, и не мешало бы время от времени проверять, а не стоил ли ее завершить.

Хорошо, но в случае с `await`, или тем же `.then` из `es6`, у вас уже есть отличная точка входа для абстракции. Вам нужно конкретно под ваш длинный загрузчик из колбеков написать `wrap` для `.then` (или `await`), который будет проверять, необходимо ли прерывать код, и если да — просто не вызывать новый `.then` (`await`).

Поправьте, если я где-то ошибся.

 Rulexec 10.12.16 в 03:51 # 📌 📄 🔄

Да, так и есть, нужно будет делать что-то вроде `.then(checkForCancel(function(data) { ... })),` где `checkForCancel` будет возвращать функцию, которая будет проверять какую-нибудь переменную, и только если она всё ещё хорошая, выполнять переданную.

 RA_ZeroTech 10.12.16 в 04:12 # 📌 📄 🔄

мне кажется, что у Вас как раз задача, решение которой заключается в обработке событий...

если я все правильно понял, то предположу, что задача похожа на «загрузить файл на сервер с возможностью отменить загрузку пока идет процесс»... Если предположение с аналогией верно, то такую задачу (правда на NodeJS), решал именно с помощью обработки событий.

 Shifty_Fox 10.12.16 в 14:27 # 📌 📄 🔄

Ага, только я был сделал `checked` не внутри `then`, а поставил бы `then` внутри `checkedThen`. Тогда у вас была бы ровно одна функция `checkes` и не надо было бы помнить что нужно писать две функции, `then` и внутри `checked`, а только одну `checkedThen`, получилось бы алгоритмически красиво :)

 Ryotsuke 10.12.16 в 08:48 # 📌 📄 🔄

Посмотрите в сторону `Observable` из `RxJS`

Их можно отменить(отписаться), они умеют генерировать несколько событий, а не строго одно — например вернуть промежуточный результат

 vladimirgalian 10.12.16 в 13:24 # 📌 📄 🔄

Этот недостаток промисов (отсутствие механизма отмены операции до завершения), а также другой — невозможность промежуточных «выстрелов» полного завершения, решается реактивным подходом. Например в `Angular2` предлагается `RxJS`, чтобы выполнить `HTTP` запрос с возможностью отмены.



stardust_kid 11.12.16 в 13:05



А что скажете по поводу метода с cancellation token?



Rulexec 11.12.16 в 22:04



Посмотрел, очень интересно, явное всегда лучше неявного. Спасибо большое, мне нравится этот подход, посмотрим как-нибудь.



raveclassic 11.12.16 в 18:42



Посмотрите в сторону генераторов и саг (в частности redux-saga). Достаточно элегантное решение обработки длинных процессов, еще и без самих сайд-эффектов как таковых.



APXEOLOG 11.12.16 в 01:41



Не понимаю, почему из коробки нет аналога async-waterfall из nodejs и приходится постоянно копипастить его самому. А еще мне не нравится, что выполнение промиса начинается при его создании. Приходится постоянно городить обертки, если работаешь с набором задач



akzhan 11.12.16 в 01:52



Эм, async поддерживает работу на стороне браузера.

Async is a utility module which provides straight-forward, powerful functions for working with asynchronous JavaScript. Although originally designed for use with Node.js and installable via `npm install --save async`, it can also be used directly in the browser.

<https://caolan.github.io/async/>



APXEOLOG 11.12.16 в 02:01



Я понимаю что его можно использовать использовать в браузере, однако я говорю о наличии данной функции из коробки. Не всегда есть возможность подключать лишние js библиотеки



alQlagin 11.12.16 в 09:30



В ноду async вам тоже нужно устанавливать и подключать. даже если нет возможности поставить пакетным менеджером всегда можно закинуть файл в папку проекта или воспользоваться cdn



num8er 11.12.16 в 11:20



поправьте пожалуйста:

```
return return resolve(1);
```

на

```
reslove(1);
```



RA_ZeroTech 11.12.16 в 11:20



спасибо. исправил



vvadzim 11.12.16 в 22:06



Есть один лайф-хак у промисов, на статью не танет)

В конструкторе промиса достаточно очень часто одного параметра — resolve. Исключение, когда reject нужно передавать куда-то как callback — в этом случае лайфхак не удобен, хотя тут на вкус.

Причина — вызов `resolve(Promise.reject(error))` эквивалентен `reject(error)`.

Возможные минусы:

- многословнее.
- создаётся лишний объект. Может быть критично, если промис возвращает ошибку слишком часто.

Возможные плюсы:

- + многословнее. Паттерн `Promise.reject()` по исходникам как-то легче ищется, и поиском и визуально.
- + меньше идентификаторов — меньше мусора в глазах. Легче реализуются вложенные конструкторы промисов (кейс редкий, но нудный, если уж столкнулся). `new Promise(resolveAaaaa =>... new Promise(resolveBbbbb =>` читается проще, нежели `new Promise((resolveAaaaa, rejectAaaaa) =>... new Promise((resolveBbbbb, resolveBbbbb) =>`.` И поиском, опять же, по `Promise.reject` легче найти все ошибки.
- + к предыдущему — проще паботать с резолверами в каких-то коллекциях, если нужно. Сохранить один-единственный callback и все операции пров


через него приятнее, нежели ныкать объект с полями {resolve, reject}. Кстати, это кейс, при котором количество создаваемых объектов обычно будет меньше, нежели с сохранением reject. При resolve(Promise.reject(...)) дополнительный объект создается только при ошибке, а при сохранении {resolve,reject} — всегда.

+ когда и если будет введён Promise.cancel(), не будет нужды добавлять в код третий параметр, достаточно будет resolve(Promise.cancel(reason)) — , же, обычно это операция, которая срабатывает редко.

 Shannon 16.12.16 в 16:23 # 🔖 📄 🔄

Не плохой вариант

↑


 iShatokhin 11.12.16 в 22:07 # 🔖

Параметр function onSuccess({}) будет вызван в случае успешного выполнения «обещания», function onFail({}) – в случае ошибки. По этой причине следующий код будет работать одинаково:
...
Гораздо привычнее и понятнее использовать catch(...).

↑

Дело не в "привычном" и "непривычном", а в том, что последовательность ловли исключений меняется. Если функция onSuccess выбросит исключе- то в onFail вы это исключение не увидите. Зато увидите в последующем catch.


Иногда ошибочно кладут внешний callback в последние then и catch одновременно (для результатов или ошибки соответственно), что приводит к двойному вызову callback, если дальше по коду будет синхронно выброшено исключение. Пример такой ошибки — <https://github.com/caolan/async/pu>

 IPri 12.12.16 в 03:41 # 🔖

Для «loadImage» можно было использовать «fetch», он как раз возвращает промис.


↑

[loadImage на основе fetch](#)

 RA_ZeroTech 12.12.16 в 12:31 # 🔖 📄 🔄

спасибо за альтернативный вариант. да, можно было бы :) это уже на усмотрение разработчика. да и нативная поддержка этого метода в браузер еще хуже, чем у самих промисов.

↑

 IPri 12.12.16 в 12:59 # 🔖

Актуальные версии всех основных браузеров (кроме Safari) уже поддерживают — [caniuse fetch](#)
Мне кажется пример с «**fetch**» с небольшим описанием был бы «изюминкой» вашей статьи. Про «fetch» всего одна статья на Хабре, в отличие от промисов.

↑

Только полноправные пользователи могут оставлять комментарии. [Войдите, пожалуйста.](#)

САМОЕ ЧИТАЕМОЕ

- Сутки
- Неделя
- Месяц

Необразованная молодёжь. Ответ бизнеса

↑ +99

👁 32,6k

🔖 95

💬 777

Самое сложное в программировании это...

↑ +67

👁 31,6k

🔖 239

💬 80

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

↑ +24

👁 15,4k

🔖 55

💬 94

[Перевод] Круглее круга: оптические эффекты при проектировании интерфейсов

↑ +78

👁 13,3k

🔖 199

💬 34

Реверс-инжиниринг первых умных часов Seiko UC-2000

↑ +136

👁 13k

🔖 78

💬 29

ИНТЕРЕСНЫЕ ПУБЛИКАЦИИ

Необразованная молодёжь: попытка подытожить и немного личного

+7 1,5k 10 6

Как работает буфер обмена в Windows

+6 1,4k 29 2

Монады для Go-программистов

+15 1,7k 23 5

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

+14 865 1 1

Семантическая разметка: LaTeX, DocBook или ???

+5 899 9 4

Аккаунт

Разделы

Информация

Услуги

Приложения

Войти

Публикации

О сайте

Реклама

Регистрация

Хабы

Правила

Тарифы

Компании

Помощь

Контент

Пользователи

Соглашение

Семинары

Песочница

Конфиденциальность

© 2006 – 2017 «TM»

Служба поддержки

Мобильная версия

