**Питання Promise**

1. Що таке Promise

це об'єкт, який може виробляти єдине значення в майбутньому: або дозволене значення, або причина, чому воно не вирішене (наприклад, сталася мережева помилка).

1. Синтаксис створення :Promise

var promise = new Promise(function(resolve, reject) {

// Эта функция будет вызвана автоматически

// В ней можно делать любые асинхронные операции,

// А когда они завершатся — нужно вызвать одно из:

// resolve(результат) при успешном выполнении

// reject(ошибка) при ошибке

})

Загальний метод обробників маркування:

promise.then(onFulfilled, onRejected)

1. setTimeout

'use strict';

// Создаётся объект promise

let promise = new Promise((resolve, reject) => {

setTimeout(() => {

// переведёт промис в состояние fulfilled с результатом "result"

resolve("result");

}, 1000);

});

// promise.then навешивает обработчики на успешный результат или ошибку

promise

.then(

result => {

// первая функция-обработчик - запустится при вызове resolve

alert("Fulfilled: " + result); // result - аргумент resolve

},

error => {

// вторая функция - запустится при вызове reject

alert("Rejected: " + error); // error - аргумент reject

}

);

В результаті запуску коду вище, "Виконав: результат" буде відображатися через 1 секунду.

1. Стани Promise

fulfilled, rejected або очікувана pending. Promise користувачі можуть прикріпити зворотні виклики для обробки завершеного значення або причини відхилення.

Обіцянки нетерплячі, а це означає, що обіцянка почне виконувати будь-яке завдання, яке ви дасте їй, як тільки буде викликана обіцянка будівельника.

1. Як працює проміс?

Обіцянка (promis) - це об'єкт, який можна повернути синхронно з асинхронної функції. Вона буде знаходитися в одному з трьох можливих станів:

* Виконано: це буде викликано (наприклад: було викликано)onFulfilled()reject()
* Відхилено: буде викликано (наприклад: було викликано)onRejected()reject()
* Очікується: ще не завершено або відхилено.

1. Важливі правила promis

Стандарт для promis був визначений спільнотою Promises/A+ Існує багато реалізацій, які відповідають стандарту, включаючи загальні вимоги стандарту ECMAScript для JavaScript.

Обіцянки наступної специфікації повинні відповідати конкретному набору правил:

Обіцянка або "тоді" - це об'єкт, який забезпечує стандартний метод..then ()

Очікувана обіцянка може перейти в виконаний або відхилений стан.

вже встановлений стан виконано - виконано або відхилено - відхилено. Не повинні і не можуть змінюватися. переїхати в інші країни.

Після встановлення держава повинна мати значення (вона може бути і). Це значення не повинно змінюватися.undefined

Зміна в цьому контексті відноситься до порівняння ідентичності (). Об'єкт можна використовувати як виконане значення, а властивості об'єкта можуть мутувати.

1. Правила методів

Метод повинен відповідати наступним правилам

* .then ()
* І , і є необов'язковими.onFulfilled ()onRejected ()
* Якщо наведені аргументи не є функціями, їх слід ігнорувати.
* onFulfilled () буде викликано після того, як обіцянка буде виконана, а значення обіцянки є першим аргументом.
* onRejected() буде викликаний після відхилення обіцянки, а причиною відмови є перший аргумент. Причиною може бути будь-яке дійсне значення JavaScript, але оскільки відхилення по суті є синонімом винятків, рекомендую використовувати об'єкти Error.
* Ні, ні не може бути викликаний більше одного разу.onFulfilled ()onRejected ()
* .then () Їх можна багато разів використовувати на одну і ту ж обіцянку. Іншими словами, обіцянка може бути використана для об'єднання зворотних вас.
* .then () Він повинен повернути нову обіцянку.promise2
* Якщо або повернути значення , a є обіцянкою, воно буде заблоковано (припустимо той самий стан і значення, що і ). В іншому випадку він буде виконаний зі значенням .onFulfilled ()onRejected ()xxpromise2xpromise2x
* Якщо або , або кидає виняток , повинні бути відхилені разом з як причина.onFulfilledonRejectedepromise2e
* Якщо вона не є функцією і виконується, вона повинна бути виконана з тим же значенням, що і .onFulfilledpromise1promise2promise1
* Якщо вона не є функцією і відхилена, вона повинна бути відхилена з тієї ж причини, що і .onRejectedpromise1promise2promise1

1. Ланцюжок promis

Саме тому, що він завжди повертає нову обіцянку, можна пов'язати обіцянки з точним контролем над тим, як і де обробляються помилки. Обіцянки дозволяють імітувати поведінку звичайного /синхронного коду.

Як і синхронний код, ланцюжок призведе до послідовності, яка працює в послідовному порядку. Іншими словами, ви можете зробити:

Припускаючи, що кожна з функцій, і повернення обіцянок, буде чекати, поки функція буде завершена перед запуском, і save() буде чекати завершення перед запуском. Спрацює тільки в тому випадку, якщо будь-яка з попередніх обіцянок буде відхилена.

1. Обробка помилок

Зверніть увагу, що обіцянки мають як успіх, так і обробник помилок, і дуже часто можна побачити код, який робить це:

Але що станеться, якщо він кине помилку? Обіцянка, повернута з , буде відхилена, але немає нічого, щоб зловити невдачу - це означає, що помилка у вашому додатку ковтається.

1. Як скасувати promis?

Одна з перших речей, які часто запитують нові користувачі, - це як скасувати promis. Ось ідея: Просто відкиньте обіцянку з "Скасувати" як причину. Якщо вам потрібно мати справу з ним інакше, ніж "звичайна" помилка, розщедритися на неї в обробнику помилки.

1. Додавання .cancel() до promis

Додавання дає promis нестандартною, але вона також порушує інше правило обіцянок: тільки функція, яка створює promis, повинна бути в змозі дозволити, відхилити або скасувати promis. Виявлення цього порушує цю інкапсуляцію і заохочує людей писати код, який маніпулює promis в місцях, які не повинні знати про це. Уникайте «спагетті» і невиконаних promis.cancel ()

1. Абстрактне відхилення обіцянок

Це добре для таймера, але ми можемо абстрагувати цю ідею далі, щоб інкапсулювати все, що вам потрібно пам'ятати: wait ()

Відкиньте обіцянку скасування дефолту - ми не хочемо скасовувати або кидати помилки, якщо обіцянка скасування не буде скасована.

Не забудьте виконати очищення, коли ви відмовляєтеся від скасування.

Пам'ятайте, що саме очищення може призвести до помилки, і ця помилка також зажадає обробки. (Зверніть увагу, що в зразку очікування немає обробки помилок - легко забути!)onCancel

1. Додаткові власні функції Promise JS

Рідна властивість Promise має деякі додаткові речі, які можуть вас зацікавити:

Promise.reject() - Повертає відхилену обіцянку.

Promise.resolve() - Повертає дозволену обіцянку.

Promise.race() - Бере масив (або будь-який ітераційний) і повертає обіцянку, яка вирішується зі значенням першої вирішеної обіцянки в ітерованій обіцянці, або відкидається через першу обіцянку, яка відкидається.

Promise.all() Бере масив (або будь-який ітераційний) і повертає обіцянку, яка вирішується, коли всі обіцянки в ітераційному аргументі вирішуються або відхиляються через першу обіцяну обіцянку, яка відкидається.

1. Що таке асинхронність /очікування?

Async/await - це відносно новий спосіб написання асинхронного (неблокуючого) коду в JS. Вони загорнуті в безлад. Це робить код більш читабельним і чистим, ніж функції promis і зворотного виклику. Однак, щоб використовувати async/await, вам потрібно добре знати програми.

ключове слово "await" можна використовувати лише в асинхронній функції. Використання функції "чекати" всередині іншої функції призведе до помилки. Очікування очікування завершення виразу праворуч, щоб повернути його значення перед виконанням наступного рядка коду.

1. Виклик відхилити з об'єктом помилки

Якщо щось піде не так, ми повинні зателефонувати. Ви можете зробити це за допомогою аргументу будь-якого типу (наприклад), але рекомендується використовувати об'єкт (або успадкований від нього). Чому це так? Скоро ми зрозуміємо.rejectresolveError

1. Негайно вирішити/відхилити виклик

Зазвичай виконавець робить щось асинхронне, а потім дзвонить /, тобто через деякий час. Але це не обов'язково, або може бути спровоковано відразу:resolverejectresolvereject

let promise = new Promise(function(resolve, reject) {

// задача, не требующая времени

resolve(123); // мгновенно выдаст результат: 123

});

Це може статися, наприклад, коли ми почали виконувати завдання, але відразу побачили, що воно вже було виконано раніше, і результат був кешований.

Ця ситуація нормальна. Ми відразу ж отримаємо успішно завершений.Promise

1. Стан і властивості результату – внутрішні

Властивості і є внутрішніми властивостями об'єкта, і ми не маємо прямого доступу до них. Для обробки результату слід використовувати методи //, про них ми поговоримо далі.stateresultPromise.then.catch.finally

1. Споживач then

Найважливішим і фундаментальним методом є..then

Синтаксис:

promise.then(

function(result) { /\* обработает успешное выполнение \*/ },

function(error) { /\* обработает ошибку \*/ }

);

Першим аргументом методу є функція, яка виконується, коли проміс переходить в стан «успішно виконаний» і отримує результат..then

Другий аргумент - це функція, яка виконується, коли проміс переходить в стан "failed" і отримує помилку..then

Наприклад, ось реакція на успішно виконаний прибуток:

let promise = new Promise(function(resolve, reject) {

setTimeout(() => resolve("done!"), 1000);

});

// resolve запустит первую функцию, переданную в .then

promise.then(

result => alert(result), // выведет "done!" через одну секунду

error => alert(error) // не будет запущена

);

Була виконана перша функція.

А в разі помилки в програмі - буде виконана друга:

let promise = new Promise(function(resolve, reject) {

setTimeout(() => reject(new Error("Whoops!")), 1000);

});

// reject запустит вторую функцию, переданную в .then

promise.then(

result => alert(result), // не будет запущена

error => alert(error) // выведет "Error: Whoops!" спустя одну секунду

);

Якщо нас цікавить тільки результат успішного виконання завдання, то на нього може бути передана тільки одна функція:then

let promise = new Promise(resolve => {

setTimeout(() => resolve("done!"), 1000);

});

promise.then(alert); // выведет "done!" спустя одну секунду

1. Catch

Якщо ми хочемо лише впоратися з помилкою, ми можемо використовувати в якості першого аргументу: . Або ви можете скористатися методом, який буде робити те ж саме:null.then(null, errorHandlingFunction).catch(errorHandlingFunction)

let promise = new Promise((resolve, reject) => {

setTimeout(() => reject(new Error("Ошибка!")), 1000);

});

// .catch(f) это то же самое, что promise.then(null, f)

promise.catch(alert); // выведет "Error: Ошибка!" спустя одну секунду

Викликом є скорочена, «укорочена» версія..catch(f).then(null, f)

1. Finally

За аналогією з блоком від звичайного, програми також мають метод .finallytry {...} catch {...}finally

Виклик схожий на , в тому сенсі, що він буде виконаний в будь-якому випадку, коли проміс завершиться: успішно або з помилкою..finally(f).then(f, f)f

finally добре підходить для очищення, наприклад, зупинки завантажувального індикатора, адже його необхідно зупинити незалежно від результату.

Наприклад:

new Promise((resolve, reject) => {

/\* сделать что-то, что займёт время, и после вызвать resolve/reject \*/

})

// выполнится, когда промис завершится, независимо от того, успешно или нет

.finally(() => остановить индикатор загрузки)

.then(result => показать результат, err => показать ошибку)

1. Відмінності:then(f,f)

* Обробник, викликаний з програми , не має аргументів. Ми не знаємо, як була завершена проміс. І це нормально, тому що зазвичай наша робота полягає в тому, щоб виконувати «загальні» фінальні процедури.finallyfinally
* Обробник «пропускає» результат або помилку далі до наступних обробників.finally

1. На завершених програмах обробники запускаються негайно

Якщо проміс знаходиться в відкладеному стані, обробники будуть чекати його. Однак, якщо проміс вже завершений, обробники будуть виконані негайно:.then/catch/finally

1. loadScript

Нова функція більше не потребує аргументу . Замість цього він створить і поверне об'єкт, який перейде в стан "успішно завершений" після завершення завантаження. Зовнішній код може додавати обробники ("абоненти") за допомогою:loadScriptcallbackPromise.then

function loadScript(src) {

return new Promise(function(resolve, reject) {

let script = document.createElement('script');

script.src = src;

script.onload = () => resolve(script);

script.onerror = () => reject(new Error(`Ошибка загрузки скрипта ${src}`));

document.head.append(script);

});

}

Застосування:

let promise = loadScript("https://cdnjs.cloudflare.com/ajax/libs/lodash.js/4.17.11/lodash.js");

promise.then(

script => alert(`${script.src} загружен!`),

error => alert(`Ошибка: ${error.message}`)

);

promise.then(script => alert('Ещё один обработчик...'));

1. Чи можна «перегрузити» проміс?

Що з'явиться в коді нижче?

let promise = new Promise(function(resolve, reject) {

resolve(1);

setTimeout(() => resolve(2), 1000);

});

promise.then(alert);

Другий виклик буде проігноровано, оскільки враховується лише перший виклик. Всі наступні виклики – ігноруються.

1. Promise після reject/resolve

Відзначимо, що після дзвінка проміс більше не може «змінити свою думку».resolve/reject

Коли проміс переходить у «виконаний» стан – з результатом (рішучістю) або помилкою (відхиленням) – це назавжди.

В результаті виклику цього коду спрацює тільки перший обробник, адже після дзвінка проміс вже отримав державу (з результатом), і в майбутньому нічого не змінить.thenresolve

1. Промисифікація

Промисифікація - це коли ви берете асинхронну функціональність і робите для неї обгортку, яка повертає проміс.

Після промікації використання функціоналу часто стає набагато зручнішим.Функція поверне проміс, який, якщо дані з успішно завантажені, перейде в ці дані, і якщо це не вдасться, він перейде з інформацією про помилку:httpGet(url)urlfulfilledrejected

1. Властивості new Promise(executor)

* PromiseState - State, спочатку "в очікуванні".
* PromiseResult є результатом, немає значення, коли ви його створюєте.
* PromiseFulfillReactions — це список функцій обробника успіху.
* PromiseRejectReactions — це список функцій обробника помилок.

1. Promise.all (ітераційний)

Виклик отримує масив (або інший ітерований об'єкт) програм і повертає проміс, який чекає завершення всіх переданих програм і переходить у стан «виконаний» з масивом їх результатів.Promise.all(iterable)

Наприклад:

Promise.all([

httpGet('/article/promise/user.json'),

httpGet('/article/promise/guest.json')

]).then(results => {

alert(results);

});

1. Promise.race(iterable)

Виклик, наприклад , отримує ітерований об'єкт з промісом, який буде виконаний, і повертає нову проміс.Promise.racePromise.all

Але, на відміну від , результатом буде тільки перший успішно виконаний прибуток зі списку. Решта ігноруються.Promise.all

Наприклад:

Promise.race([

httpGet('/article/promise/user.json'),

httpGet('/article/promise/guest.json')

]).then(firstResult => {

firstResult = JSON.parse(firstResult);

alert( firstResult.name ); // iliakan або guest, що раніше завантажиця

});

1. Promise.resolve(value)

Виклик створює успішно виконану програму з результатом .Promise.resolve(value)value

Він схожий на дизайн:

new Promise((resolve) => resolve(value))

Promise.resolve використовувати, коли вони хочуть побудувати асинхронний ланцюжок, і початковий результат вже є.

Наприклад:

Promise.resolve(window.location) // почати з цього значення

.then(httpGet) // викликати для нього httpGet

.then(alert) // вивести результат

1. Promise.reject(error)

Аналогічно створює вже виконану проміс, але не з успішним результатом, а з помилкою.Promise.reject(error)error

Наприклад:

Promise.reject(new Error("..."))

.catch(alert) // Error: ...

Метод використовується дуже рідко, набагато рідше, ніж , тому що помилка зазвичай виникає не на початку ланцюжка, а в процесі його виконання.Promise.rejectresolve