¿Qué es una promesa en Javascript?

Formular una pregunta

70

28

Partiendo de:

> No prometas aquello que no puedes cumplir

Podría suponer que una promesa en Javascript se basa en este principio asegurando que siempre habrá un resultado (*¿esperado?*).

Teniendo de ejemplo este código:

```
var promesa = new Promise(
    function(resolve, reject) {
        alert('Hola mundo!');
    }
);

promesa.then(
    function(value) {
        alert('Hola universo!');
}).then(
    function(value) {
        alert('Hola multiuniverso!');
});
```

Se desencadena un proceso en *serie* hasta mostrar `alert('Hola multiuniverso!');` , lo cual también se podría hacer con *Ajax Requests* anidados, lo cual me lleva a suponer que también existen otras ventajas contra una simple llamada Ajax.

Las principales cuestiones serían (ya sé que son varias y no es tan bien visto dentro de SOes, pero con la explicación de lo que es una promesa se resuelven varias de ellas):

- ¿Qué es una promesa?

- ¿Ajax es un tipo de promesa?
- ¿Las promesas pueden ser llamadas síncronas y asíncronas?

javascript promesas

formulada el 20 abr. 17 a las 17:44

Phi
8,418 3 19 44

15 `var promesa = new esSOPromesa ("En cuanto pasen las 48 hrs pondré una jugosa recompensa")` Eso es una **promesa** es esSO... :) – A. Cedano el 20 abr. 17 a las 17:46

3 Esta pregunta y la respuesta que publiqué puede darte una idea: es.stackoverflow.com/q/2799/822 – fredyfx el 21 abr. 17 a las 18:12

2 respuestas

79

+100

> [...] lo cual también se podría hacer con Ajax Requests anidados, lo cual me lleva a suponer que también existen otras ventajas contra una simple llamada Ajax.

Como introducción, te diré que las promesas van mucho más allá de peticiones AJAX. No tienen relación directa, salvo por el concepto de *asincronía* que comparten.

¿Qué es una promesa?

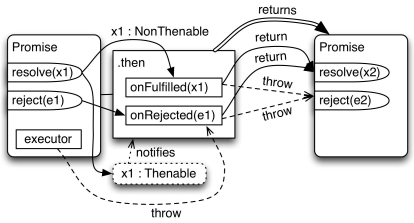
Mozilla nos da un concepto bastante claro y conciso:

> El objeto `Promise` (Promesa) es usado para computaciones asíncronas. Una promesa representa un valor que puede estar disponible ahora, en el futuro, o nunca.

Una promesa, como su propio nombre lo dice, es simplemente un objeto que puede o no devolver algún valor en la línea de tiempo presente y futuro. Me gusta describir una promesa como una especie de

> Tú haces algo, y en consecuencia obtendrás algo, ahora o en un futuro.

Esto aplica igual a las promesas, tu ejecutas código asíncrono y obtienes la promesa de que obtendrás una respuesta, que puede ser en ese instante o en un futuro.



Ciclo de vida de una promesa

La historia de las promesas se remonta al especificación Promises/A+ que detalla cómo cualquier implementación *compilante* debe implementarlas.

Una Promesa *recuerda* el contexto en donde se ejecuta, es decir, sabe con precisión en qué punto se ha de resolver un valor o lanzar un error. Cuando una promesa entra en ejecución pasa a tener 2 estados, uno inicial y uno final:

- pending (pendiente)
- fullfilled (resuelta exitosamente)
- rejected (rechazada)

Inicialmente, una promesa tiene el estado *pending*, estado que tendrá hasta que la promesa haya resuelto un valor mediante `resolve` o haya ocurrido un error (`reject`). Cuando una promesa alcanza uno de estos dos estados, ya no puede realizar transiciones a otro.

En términos generales, Promises ES6 implementa la especificación Promises/A+ casi de igual forma. Sin embargo, a diferencia de la especificación Promises/A+, en donde no se habla de un método especial para capturar errores, la implementación de ECMAScript añadió el método `catch` que resolverá cualquier error que haya ocurrido durante la ejecución del código.

Vendría a ser como una especie de *arquitectura*. El concepto de Promesas está ligado a la programación concurrente. Tanto Promesas, como Delay, Future (bien conocido en Java) y Deferred, estos actúan como un *proxy* para un resultado que, inicialmente desconocemos.

No es un plugin

Promise es una especificación, no un plugin; aunque existen varios plugins para usar; el más conocido es `Q` .

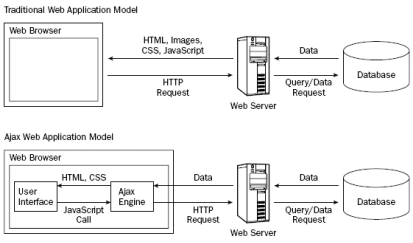
Sí es un estándar

De hecho, la especificación de Promises se remonta más allá de ES2015. Promises es una especificación conocida como Promses/A+ la cual fue implementada en ECMAScript 2015/6, por lo tanto, esta especificación pertenece al estándar ECMAScript que actualmente es implementada al 98% en la mayoría de navegadores modernos (Safari Tech Preview es el único navegador en implementar ES6 al 100%).

Puedes ver las propuestas actuales en el repositorio del comité.

¿Ajax es un tipo de promesa?

No es un tipo de promesa, es una especificación. AJAX, de las siglas **Asynchronous JavaScript And XML** es una técnica para realizar peticiones asíncronas; cuyo principal efecto es obtener una respuesta en un tiempo indeterminado pero sin afectar el proceso principal, lo cual evita una recarga del documento actual. La historia de AJAX se remonta a finales de los 90's, durante el desarrollo de Microsoft Outlook.



Cómo funciona AJAX

W3.

Como dije en la introducción, AJAX no tiene que ver directamente con Promesas porque no la implementan, al menos no `XMLHttpRequest`. Por otro lado, jQuery implementa su propio sistema *deferred* que se usa en `$.ajax` o directamente mediante `$.deferred`. Sin embargo, hace poco salió a la luz una nueva API para AJAX: `fetch` que sí usa Promesas nativamente.

¿Las promesas pueden ser llamadas síncronas y asíncronas?

No. Una promesa **siempre será asíncrona** y no hay forma de hacerla síncrona. Lo que se puede hacer es *hacerla ver* como si fuese síncrona y esto es gracias a `Async/Await` que acaba de alcanzar el stage 4 y será incluída en la versión ECMAScript de éste año. Usando ésta nueva especificación podemos *esperar* por la respuesta sin que deje de ser asíncrona, como si fuese una función síncrona estándar.

Ejemplo (se necesita Chromium 55+ para ser ejecutado)

```
function divide(dividendo, divisor)
  return new Promise((resolve, reje
    if (divisor === 0) {
      reject(new Error('No se puede
    } else {
      resolve(dividendo/divisor);
    }
  });
}

async function test() {
  // esperamos por la respuesta
  try {
    const result = await divide(5,
    console.log(result);
  } catch (err) {
    console.error(err.message);
  }
}

test();
```

Ejecutar Ampliar

Como se puede observar, a simple vista pareciese que ejecutamos la

escena. Cuando usamos `await` lo que hacemos es *esperar* por la respuesta de aquella promesa. Técnicamente, se está pausando la ejecución -sin bloquear- del código hasta que la promesa sea resuelta (fullfilled o rejected).

Una vez que la promesa ha sido resuelta, la ejecución del código seguirá. Si necesitas entender a detalle cómo funciona la VM y el EventLoop, te recomiendo mucho ver este vídeo de la conferencia JSFOO del año ante pasado.

¿Debería usar Promesas?

Mi respuesta es **absolutamente sí**. Úsalas siempre que puedas, mejora semánticamente y es una mejora cognitiva también (si has visto un callback hell te da un derrame cerebral), además, en conjunto con las funciones asíncronas (`async/await`) puedes tener un código limpio, ordenado y sobre todo asíncrono; ¡Lo que Python logró con asyncio!.

Sin embargo, no debes desechar los callbacks porque son pieza fundamental en un lenguaje funcional. **Las promesas no llegaron para reemplazar a los callbacks**, sino, como un complemento al lenguaje. En términos generales, usar promesas nos provee de un mayor control del código asíncrono gracias a particularidades que tienen y que carecen los callbacks (como chaining).

Algunos pros:

- Mejor control de funciones asíncronas
- Integración con nuevas APIs
- Se puede usar como wrapper para callbacks si se necesita refactorizar
- Alineación con el estándar `async/await`

Fuentes:

- Mozilla Developer Network

jasilva
4,262 4 22 41

respondida el 21 abr. 17 a las 1:30

gugadev
16.6k 18 41

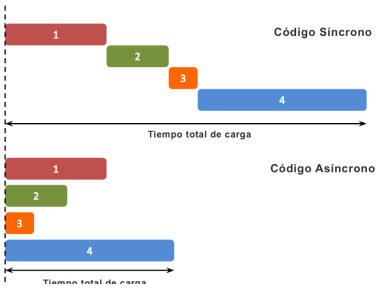
5 Genial respuesta. Solo quería comentar la frase: `Lo que se puede hacer es hacerla ver como si fuese síncrona y esto es gracias a Async/Await` . Yo diría que más que de forma síncrona, async/await permite escribir código asíncrono de manera "secuencial". – Jose Hermosilla Rodrigo el 21 abr. 17 a las 18:16

Gracias por comentar, @JoseHermosillaRodrigo. Por lo general, una ejecución síncrona quiere decir "bloqueante", y "bloqueante" quiere decir que el hilo de espera a que termine dicha ejecución, por lo que la lectura del código será secuencial. – gugadev el 21 abr. 17 a las 18:24

27

Entendiendo las promesas y su importancia

Hay algo fundamental para poder entender las promesas y la revolución que suponen. JavaScript es de un solo hilo, es decir, dos porciones de secuencia de comandos no se pueden ejecutar al mismo tiempo, tienen que ejecutarse uno después del otro. En navegadores, JavaScript comparte un hilo con una carga de otras cosas que difiere de navegador en navegador. Pero, generalmente, JavaScript se encuentra en la misma cola que la pintura, la actualización de estilos y el control de acciones de usuario (como destacar texto e interactuar con controles de formulario). La actividad en uno de estos elementos retarda a los otros.



Para evitar eso, hasta ahora se han usado eventos y callbacks.

Por ejemplo:

```
var img1 = document.querySelector('.i
img1.addEventListener('load', functio
  // imagen cargada
});

img1.addEventListener('error', functi
  // algo salió mal
});
```

Por desgracia, en el ejemplo anterior, es posible que los eventos ocurran antes de que comencemos a escucharlos. Por eso, debemos solucionar este problema usando la propiedad “complete” de las imágenes:

```
var img1 = document.querySelector('.i

function loaded() {
  // imagen cargada
}

if (img1.complete) {
  loaded();
}
else {
  img1.addEventListener('load', loade
}

img1.addEventListener('error', functi
  // algo salió mal
});
```

Esto no captura imágenes que generaron n error antes de que pudiéramos escucharlas. Lamentablemente, el DOM no nos brinda una forma de hacerlo. Además, en este ejemplo, solo intentamos cargar una imagen. La complejidad aumenta aún más cuando deseamos saber cuándo se cargó un conjunto de imágenes.

Los eventos son excelentes para cosas que pueden suceder varias veces en el mismo objeto, porque en ese caso no interesa saber realmente lo que ocurrió antes de adjuntar el receptor. Pero si se trata de éxito/fallo asincrónico, idealmente, querrás algo así:

```
img1.callThisIfLoadedOrWhenLoaded(fun
  // cargada
}).orIfFailedCallThis(function() {
  // fallo
});
```

```
  // uno o más fallos
});
```

Las promesas hacen eso, aunque con una mejor nomenclatura. Si los elementos de imagen HTML tuviesen un método "ready" que mostrara una promesa, podríamos hacer lo siguiente:

```
img1.ready().then(function() {
  // éxito
}, function() {
  // fallo
});

// y…
Promise.all([img1.ready(), img2.ready
  // todo bien
}, function() {
  // al menos un fallo
});
```

Fundamentalmente, las promesas se parecen un poco a los receptores de eventos, a excepción de lo siguiente:

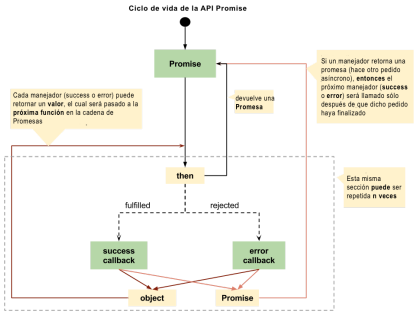
- Una promesa solo puede completarse con éxito o fallar una vez. No puede completarse con éxito o fallar dos veces, ni puede pasar de exitoso a fallido ni viceversa.
- Si una promesa se ha completado con éxito o ha fallado y luego agregas un callback de exitoso/fallido, se llamará al callback correcto, a pesar de que el evento haya sucedido antes.

Esto es extremadamente útil para el éxito o fracaso de procesos asincrónicos porque es menos importante el momento exacto de la disponibilidad que la reacción ante el resultado.

Entonces... ¿qué son las promesas?

Son una API que nos ayudará a realizar cosas antes complicadas o imposibles debido a lo que se ha dicho más arriba.

La imagen nos muestra el *ciclo de vida* y el funcionamiento de una promesa. Como *prometido*, he traducido las explicaciones de la imagen :)



Una promesa puede ser de estas clases:

1. `fulfilled` (cumplida): la acción relacionada con la promesa se completa con éxito.
2. `rejected` (rechazada): la acción relacionada con la promesa no se completa con éxito.
3. `pending` (pendiente): aún no se completa ni se rechaza.
4. `settled` (finalizada): se completa o se rechaza.

En las especificaciones, también aparece el término `thenable` para describir un objeto parecido a una promesa porque tiene un método `then`.

Cabe decir que hace tiempo que las promesas existen en forma de bibliotecas. Las siguientes son algunas:

- Q
- when
- WinJS
- RSVP.js

La revolución de las promesas empezó a través de librerías como estas y de frameworks que las usan como medio primario para manipular la asincronía en su código. Desde el 2013, las promesas están disponibles de manera nativa en los exploradores modernos, lo cual será decisivo en el futuro.

Estas bibliotecas y las promesas de JavaScript tienen en común un comportamiento estandarizado llamado Promises/A+. Si usas jQuery, encontrarás algo similar llamado Deferred. Sin embargo, Deferred no cumple con Promise/A+, por lo cual es un tanto diferente y menos útil, así

trata de un subconjunto de Deferred y no funciona muy bien.

Si bien las implementaciones de las promesas cumplen con un comportamiento estandarizado, las API generales son diferentes. Las API de las promesas de JavaScript son similares a las de RSVP.js.

Las promesas de JavaScript empezaron en DOM como “Future”, se les cambió el nombre a “Promise” y, finalmente, se trasladaron a JavaScript. Es fabuloso contar con ellas en lugar del DOM en JavaScript porque estarán disponibles en contextos de JS sin navegador, como Node.js.

Si bien son una funcionalidad de JavaScript, el DOM las usa sin problemas cuando las necesita. De hecho, todas las nuevas API de DOM con métodos de éxito o falla asincrónicos usan promesas.

Viendo una promesa por dentro

Una promesa se crea así:

```
var promise = new Promise(function(re
  // hacer algo que puede ser asíncro

  if (/* todo está bien */) {
    resolve("Exito");
  }
  else {
    reject(Error("Algo falló"));
  }
});
```

El constructor de la promesa recibe un argumento: un callback con dos parámetros (resolve y reject). A continuación, se hace algo con el callback (tal vez un proceso asincrónico) y se llama a resolve si todo funciona bien o a reject si esto no sucede.

Como en throw del JavaScript que todos conocemos, es costumbre (aunque no obligación) aplicar reject con un objeto Error. La ventaja de los objetos Error es que capturan un seguimiento de pila; de esta forma, las herramientas de depuración son más útiles.

Para usar esta promesa:

```
  console.log(err); // Error: "Hubo u
});
```

`then()` recibe dos argumentos: un callback para cuando se tiene éxito y otro para cuando sucede lo contrario. Ambos son opcionales; puedes agregar un callback solo para cuando se tiene éxito o se produce una falla.

Uso básico de las Promesas

Pienso esta parte como una especie de *Promise by the example*, para mostrar algunos casos y ejemplos de uso de las promesas.

El constructor `new Promise()` sólo debe utilizarse para tareas asíncronas heredadas, como el uso de `setTimeout` o `XMLHttpRequest` . Se crea una nueva promesa con la nueva palabra clave y la promesa proporciona funciones de resolución y rechazo a la devolución de llamada proporcionada:

```
var p = new Promise(function(resolve,

    // Hacer tarea asíncrona y then..

    if(/* éxito */) {
        resolve('Success!');
    }
    else {
        reject('Fallo!');
    }
});

p.then(function() {
    /* hacer algo con el resultado */
}).catch(function() {
    /* error :( */
})
```

Corresponde al desarrollador llamar manualmente a `resolve` o `reject` dentro del cuerpo de la devolución de llamada en función del resultado de su tarea. Un ejemplo realista sería convertir `XMLHttpRequest` a una tarea basada en la promesa:

```
function get(url) {
  // Devolver una nueva promesa.
  return new Promise(function(resolve
    // Haz lo habitual de XHR
    var req = new XMLHttpRequest();
    req.open('GET', url);

    req.onload = function() {
      // Esto es llamado incluso en e
      // entonces chequea el status
      if (req.status == 200) {
        // Resuelve la promesa con la
```

```
        reject(Error(req.statusText))
      }
    };

    // Manejar errores de red
    req.onerror = function() {
      reject(Error("Error de Red"));
    };

    // Make the request
    req.send();
  });
}

// Use esto!
get('story.json').then(function(respo
  console.log("Éxito!", response);
}, function(error) {
  console.error("Fallo!", error);
});
```

A veces no es necesario completar tareas asíncronas dentro de la promesa, si es posible que se tome una acción asíncrona, sin embargo, devolver una promesa será lo mejor para que siempre pueda contar con una promesa que sale de una función dada . En ese caso, simplemente puede llamar a `promise.resolve()` o `promise.reject()` sin usar la nueva palabra clave. Por ejemplo:

```
var userCache = {};

function getUserDetail(username) {
// En ambos casos, en caché o no, se

  if (userCache[username]) {
    // Retorna una promise sin la pal
    return Promise.resolve(userCache[
  }

  // Use la API fetch API para obtene
  // fetch devuelve una promise
  return fetch('users/' + username +
    .then(function(result) {
      userCache[username] = result;
      return result;
    })
    .catch(function() {
      throw new Error('Usuario no enc
    });
}
```

Puesto que siempre se devuelve una promesa, siempre puede usar los métodos `then` y `catch` en su valor de retorno.

then

Todas las instancias de Promise tienen un método `then` que nos permite reaccionar a la promesa. El primer método de devolución de llamada recibe el resultado dado por la llamada `resolve()` :

```
})
.then(function(result) {
    console.log(result);
});

// En la consola:
// 10
```

La llamada de retorno se activa cuando se resuelve la promesa. También puede encadenar las devoluciones de llamada del método:

```
new Promise(function(resolve, reject)
    // Una tarea ansíncrona usando se
    setTimeout(function() { resolve(1
})
.then(function(num) { console.log('fi
.then(function(num) { console.log('se
.then(function(num) { console.log('la

// En la consola:
// first then:  10
// second then:  20
// last then:  40
```

Cada `then` recibe el resultado del valor de retorno anterior.

Si una promesa ya se ha resuelto pero se vuelve a llamar, la devolución de llamada se disparará inmediatamente. Si la promesa es rechazada y usted llama entonces después del rechazo, el callback nunca se llama.

El callback `catch` se ejecuta cuando se rechaza la promesa:

```
new Promise(function(resolve, reject)
    // Una tarea asíncrona usando set
    setTimeout(function() { reject('D
})
.then(function(e) { console.log('done
.catch(function(e) { console.log('cat

// From the console:
// 'catch: Done!'
```

Lo que usted proporcione al método de rechazo depende de usted. Un patrón frecuente es enviar un error a la captura:

```
reject(Error('Data could not be found
```

promise.all

Piense en los cargadores de JavaScript: hay momentos en los que se desencadenan múltiples interacciones asíncronas, pero sólo se quiere responder cuando se completan todos ellos - ahí es donde

devolución de llamada una vez todos están resueltos:

```
Promise.all([promise1, promise2]).the
    // Ambas promesas resultas
})
.catch(function(error) {
    // Una o más promesas rechazadas
});
```

Una forma perfecta de pensar en `Promise.all` es disparar múltiples solicitudes AJAX (via fetch) al mismo tiempo:

```
var request1 = fetch('/users.json');
var request2 = fetch('/articles.json'

Promise.all([request1, request2]).the
    // Todas las promesas resueltas!
});
```

Podrías combinar APIs como `fetch` y `Battery API`, ya que ambas retornan promesa:

```
Promise.all([fetch('/users.json'), na
    //  Todas las promesas resueltas!
});
```

Lidiar con el rechazo es, por supuesto, difícil. Si alguna promesa es rechazada el `catch` es lanzado en el primer rechazo:

```
var req1 = new Promise(function(resol
    // Una tarea asíncrona usando set
    setTimeout(function() { resolve('
});
var req2 = new Promise(function(resol
    // A mock async action using setT
    setTimeout(function() { reject('S
});
Promise.all([req1, req2]).then(functi
    console.log('Then: ', results);
}).catch(function(err) {
    console.log('Catch: ', err);
});

// From the console:
// Catch: Second!
```

`Promise.all` será super útil a medida que más APIs se mueven hacia promesas.

Promise.race

```
`Promise.race` es una función interes
sean resueltas o rechazadas, `Promise
rechaza cualquier promesa en la matri

var req1 = new Promise(function(resol
    // Tarea asíncrona usando setTime
    setTimeout(function() { resolve('
```

```
Promise.race([req1, req2]).then(funct
    console.log('Then: ', one);
}).catch(function(one, two) {
    console.log('Catch: ', one);
});

// Consola
// Then: Second!
```

Un caso de uso podría estar provocando una solicitud a una fuente primaria y una fuente secundaria (en caso de que la primaria o la secundaria no estén disponibles).

Compatibilidad con navegadores y polyfill

En la actualidad, ya existen implementaciones de promesas en los navegadores.

A partir de Chrome 32, Opera 19, Firefox 29, Safari 8 y Microsoft Edge, las promesas vienen habilitadas de forma predeterminada.

Consulta el polyfill si deseas que los navegadores sin implementaciones completas de promesas cumplan con las especificaciones, o si quieres agregar promesas a otros navegadores y Node.js.

Compatibilidad con otras bibliotecas

La API de las promesas de JavaScript tratará a todos los elementos con un método `then()` como si fueran promesas (o thenable, si se usa el idioma de las promesas). Por lo tanto, no habrá problema si usas una biblioteca que muestra promesas; funcionará bien con las nuevas promesas de JavaScript.

A pesar de que, como ha dicho, los Deferreds de jQuery son un poco inútiles. Afortunadamente, puedes transmitirlos a las promesas convencionales. Vale la pena hacerlo lo más pronto posible.

Ejemplo:

```
var jsPromise = Promise.resolve($.aja
```

En este caso, `$.ajax` de jQuery muestra un elemento Deferred. Ya que tiene un método `then()`,

varios argumentos a sus callbacks, por ejemplo:

```
var jqDeferred = $.ajax('/whatever.js

jqDeferred.then(function(response, st
  // ...
}, function(xhrObj, textStatus, err)
  // ...
})
```

En cambio, las promesas de JS ignoran todos menos el primero:

```
jsPromise.then(function(response) {
  // ...
}, function(xhrObj) {
  // ...
})
```

Afortunadamente, esto suele ser lo que quieres o, al menos, te brinda acceso a lo que quieres. Además, ten en cuenta que jQuery no sigue la convención de pasar objetos Error a rechazos.

Conclusión

Las promesas han sido un tema candente para los últimos años, y han pasado de un patrón de framework de JavaScript a un **elemento básico del idioma**. Iremos viendo cómo la mayoría de las nuevas API JavaScript se implementarán con un patrón basado en la promesa ...

... y eso es una gran cosa! Gracias a las promesas, los desarrolladores serán capaces de evitar el infierno de devolución de llamada (callback) y las interacciones asíncronas pueden ser transmitidas como cualquier otra variable. Quizá tome un poco de tiempo acostumbrarse a usarlas, pero ya tenemos a mano las herramientas, pues son nativas en la mayoría de navegadores modernos. ¡Ahora es el tiempo de aprender a usarlas!

Enlaces

- Especificación de la API Promises
- El objeto Promise: MDN
- Introducción muy buena, con un ejemplo concreto de la vida real
- Promises for Dummyes:

editada el 30 abr. 17 a las 20:39

respondida el 23 abr. 17 a las 5:24

A. Cedano

48.5k 7 60 133