INTRODUCCION A PROMESAS & ASYNC/AWAIT

2º DAW - Desarrollo Web en Entorno Cliente

Introducción: callbacks

- Muchas funciones son proporcionadas por el entorno de host de Javascript que permiten programar acciones asíncronas. En otras palabras, acciones que iniciamos ahora, pero que terminan más tarde. Por ejemplo, una de esas funciones es la función ***setTimeout***.

- Hay otros ejemplos del mundo real de acciones asincrónicas, p. ej.: la carga de scripts y módulos (a cubrirse en capítulos posteriores). Echa un vistazo a la función loadScript(src), que carga un código script src dado:

```
function loadScript(src) {
  // crea una etiqueta <script> y la agrega a la página
  // esto hace que el script dado: src comience a cargarse y ejecutarse cuando se com
  let script = document.createElement('script');
  script.src = src;
  document.head.append(script);
}
```

Esto inserta en el documento una etiqueta nueva, creada dinámicamente, <script src =" ... "> con el código src dado. El navegador comienza a cargarlo automáticamente y lo ejecuta cuando la carga se completa. Podemos usar esta función así:

```
1  // cargar y ejecutar el script en la ruta dada
2  loadScript('/my/script.js');
```

El script se ejecuta “asincrónicamente”, ya que comienza a cargarse ahora, pero se ejecuta más tarde, cuando la función ya ha finalizado. El código debajo de loadScript (...), no espera que finalice la carga del script.

```
1  loadScript('/my/script.js');
2  // el código debajo de loadScript
3  // no espera a que finalice la carga del script
4  // ...
```

Digamos que necesitamos usar el nuevo script tan pronto como se cargue. Este script declara nuevas funciones, y las queremos ejecutar. Si lo hacemos inmediatamente después de llamar a loadScript (...), no funcionarán:

```
1  loadScript('/my/script.js'); // el script tiene a "function newFunction() {…}"
2
3  newFunction(); // no existe dicha función!
```

Es natural, porque el navegador no tuvo tiempo de cargar el script. Hasta el momento, la función loadScript no proporciona una forma de monitorear la finalización de la carga. El script se carga y finalmente se ejecuta, eso es todo. Pero necesitamos saber cuándo sucede, para poder usar las funciones y variables nuevas de dicho script.

Agreguemos a loadScript un segundo argumento: una función callback que se ejecuta cuando se completa la carga el script:

```
function loadScript(src, callback) {
  let script = document.createElement('script');
  script.src = src;

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

  document.head.append(script);
}
```

El evento onload, que se describe en el artículo Carga de recursos: onload y onerror, básicamente ejecuta una función después de que el script fue cargado y ejecutado.

Ahora, si queremos llamar las nuevas funciones desde el script, lo hacemos dentro de la callback:

```
loadScript('/my/script.js', function() {
  // la callback se ejecuta luego que se carga el script
  newFunction(); // ahora funciona
  ...
});
```

Esa es la idea: el segundo argumento es una función (generalmente anónima) que se ejecuta cuando se completa la acción. Aquí un ejemplo ejecutable con un script real:

```
function loadScript(src, callback) {
  let script = document.createElement('script');
  script.src = src;
  script.onload = () => callback(script);
  document.head.append(script);
}

loadScript('https://cdnjs.cloudflare.com/ajax/libs/lodash.js/3.2.0/lodash.js', script => {
  alert(`Genial, el script ${script.src} está cargado`);
  alert( _ ); // _ es una función declarada en el script cargado
});
```

Eso se llama programación asincrónica “basado en callback”. Una función que hace algo de forma asincrónica debería aceptar un argumento de callback donde ponemos la función por ejecutar después de que se complete. Aquí lo hicimos en loadScript, pero por supuesto es un enfoque general.

Callback en una callback

¿Cómo podemos cargar dos scripts secuencialmente, el segundo en cuanto haya terminado de cargarse el primero? La solución natural sería poner la segunda llamada loadScript dentro de la callback, así:

```
loadScript('/my/script.js', function(script) {

  alert(`Genial, el ${script.src} está cargado, carguemos uno más`);

  loadScript('/my/script2.js', function(script) {
    alert(`Genial, el segundo script está cargado`);
  });

});
```

Una vez que se completa el loadScript externo, la callback inicia el interno. ¿Qué pasa si queremos un script más ...?

```
loadScript('/my/script.js', function(script) {

  loadScript('/my/script2.js', function(script) {

    loadScript('/my/script3.js', function(script) {
      // ...continúa después que se han cargado todos los scripts
    });

  });

});
```

Entonces, cada nueva acción está dentro de una callback. Esto es adecuado para algunas acciones, pero no en todos los casos; así que pronto veremos otras variantes.

Manejo de errores

En los ejemplos anteriores no consideramos los errores. ¿Qué pasa si falla la carga del script? Nuestra callback debería poder reaccionar ante eso. Aquí una versión mejorada de loadScript que monitorea los errores de carga:

```
function loadScript(src, callback) {
  let script = document.createElement('script');
  script.src = src;

  script.onload = () => callback(null, script);
  script.onerror = () => callback(new Error(`Error de carga de script con ${src}`));

  document.head.append(script);
}
```

Para una carga exitosa llama a callback(null, script) y de lo contrario a callback(error). El uso:

```
loadScript('/my/script.js', function(error, script) {
  if (error) {
    // maneja el error
  } else {
    // script cargado satisfactoriamente
  }
});
```

Una vez más, la receta que usamos para loadScript es bastante común. Es un estilo que se conoce como “error first callback” (callback con el error primero). La convención es:

1. El primer argumento de la ‘callback’ está reservado para un error, si este ocurre. En tal caso se llama a callback(err).
2. El segundo argumento (y los siguientes si es necesario) son para el resultado exitoso. En este caso se llama a callback(null, result1, result2 ...).

Así usamos una única función de ‘callback’ tanto para informar errores como para transferir resultados.

Pirámide infernal

A primera vista, es una forma viable de codificación asincrónica. Y de hecho lo es. Para una o quizás dos llamadas anidadas, se ve bien. Pero para múltiples acciones asincrónicas que van una tras otra, tendremos un código como este:

```
loadScript('1.js', function(error, script) {

  if (error) {
    handleError(error);
  } else {
    // ...
    loadScript('2.js', function(error, script) {
      if (error) {
        handleError(error);
      } else {
        // ...
        loadScript('3.js', function(error, script) {
          if (error) {
            handleError(error);
          } else {
            // ...continua después de que se han cargado todos los script (*)
          }
        });

      }
    });
  }
});
```

En el código de arriba:

- Cargamos 1.js, entonces si no hay error…
- Cargamos 2.js, entonces si no hay error…
- Cargamos 3.js, entonces, si no hay ningún error: haga otra cosa (*).

A medida que las llamadas se anidan más, el código se vuelve más profundo y difícil de administrar, especialmente si tenemos un código real en lugar de '…' que puede incluir más bucles, declaraciones condicionales, etc. A esto se le llama "infierno de callbacks" o "pirámide infernal" ("callback hell", "pyramid of doom").

La "pirámide" de llamadas anidadas crece hacia la derecha con cada acción asincrónica. Pronto se sale de control.

Entonces esta forma de codificación no es tan buena. Podemos tratar de aliviar el problema haciendo, para cada acción, una función independiente:

```
loadScript('1.js', step1);

function step1(error, script) {
  if (error) {
    handleError(error);
  } else {
    // ...
    loadScript('2.js', step2);
  }
}

function step2(error, script) {
  if (error) {
    handleError(error);
  } else {
    // ...
    loadScript('3.js', step3);
  }
}

function step3(error, script) {
  if (error) {
    handleError(error);
  } else {
    // ...continua después de que se han cargado todos los scripts (*)
  }
}
```

¿Lo Ves? Hace lo mismo, y ahora no hay anidamiento profundo porque convertimos cada acción en una función de nivel superior separada.

Funciona, pero el código parece una hoja de cálculo desgarrada. Es difícil de leer, y habrás notado que hay que saltar de un lado a otro mientras lees. Es un inconveniente, especialmente si el lector no está familiarizado con el código y no sabe dónde dirigir la mirada. Además, las funciones llamadas step* son de un solo uso, existen únicamente para evitar la “Pirámide de callbacks”. Nadie los reutilizará fuera de la cadena de acción. Así que hay muchos nombres abarrotados aquí.

Promesas

- Un "código productor" que hace algo y toma tiempo. Por ejemplo, algún código que carga los datos a través de una red. Ejemplo de paralelismo → Eso es un "cantante".
- Un "código consumidor" que quiere el resultado del "código productor" una vez que está listo. Muchas funciones pueden necesitar ese resultado. Ejemplo de paralelismo → Estos son los "fans".
- Una promesa es un objeto JavaScript especial que une el "código productor" y el "código consumidor". En términos de nuestra analogía, esta es la "lista de suscripción". El "código productor" toma el tiempo que sea necesario para producir el resultado prometido, y la "promesa" hace que ese resultado esté disponible para todo el código suscrito cuando esté listo.
- La analogía no es terriblemente precisa, porque las promesas de JavaScript son más complejas que una simple lista de suscripción: tienen características y limitaciones adicionales. Pero está bien para empezar. La sintaxis del constructor para un objeto promesa es:

```
1  let promise = new Promise(function(resolve, reject) {
2    // Ejecutor (el código productor, "cantante")
3  });
```

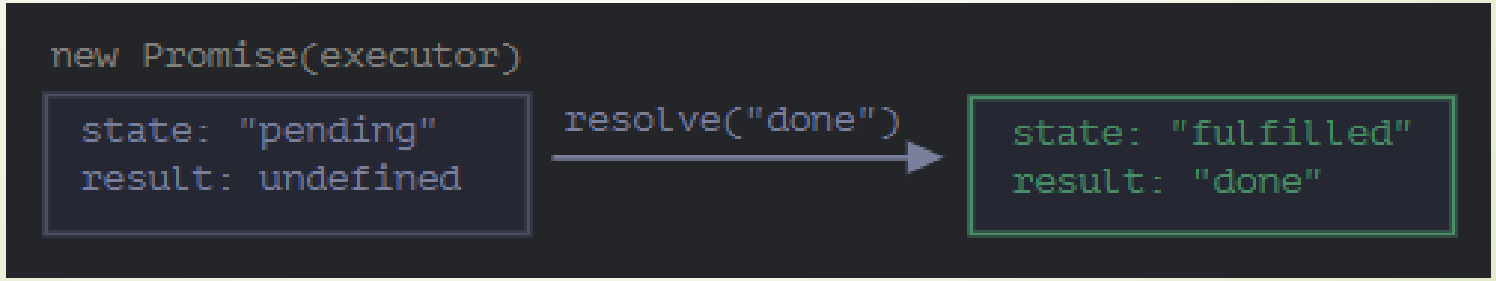
Más adelante veremos cómo los “fanáticos” pueden suscribirse a estos cambios.

Aquí hay un ejemplo de un constructor de promesas y una función ejecutora simple con “código productor” que toma tiempo (a través de setTimeout):

```
let promise = new Promise(function(resolve, reject) {
  // la función se ejecuta automáticamente cuando se construye la promesa

  // después de 1 segundo, indica que la tarea está hecha con el resultado "hecho"
  setTimeout(() => resolve("hecho"), 1000);
});
```

Podemos ver dos cosas al ejecutar el código anterior:

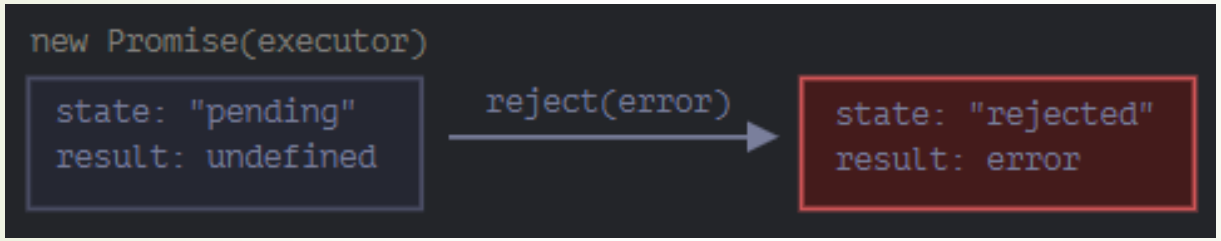
1. Se llama al ejecutor de forma automática e inmediata (por new Promise).
2. El ejecutor recibe dos argumentos: resolve y reject. Estas funciones están predefinidas por el motor de JavaScript, por lo que no necesitamos crearlas. Solo debemos llamar a una de ellas cuando esté listo.
   Después de un segundo de “procesamiento”, el ejecutor llama a resolve("hecho") para producir el resultado. Esto cambia el estado del objeto promise:



Ese fue un ejemplo de finalización exitosa de la tarea, una “promesa cumplida”. Y ahora un ejemplo del ejecutor rechazando la promesa con un error:

```
let promise = new Promise(function(resolve, reject) {
  // después de 1 segundo, indica que la tarea ha finalizado con un error
  setTimeout(() => reject(new Error("¡Vaya!")), 1000);
});
```

La llamada a reject(...) mueve el objeto promise al estado "rechazado":



Para resumir, el ejecutor debe realizar una tarea (generalmente algo que toma tiempo) y luego llamar a “resolve” o a “reject” para cambiar el estado del objeto promise correspondiente.

El estado inicial de una promesa es “pendiente”. En cuanto se resuelve o rechaza, la consideramos “establecida”.

Consumidores: then y catch

Un objeto Promise sirve como enlace entre el ejecutor (el "código productor" o el "cantante") y las funciones consumidoras (los "fanáticos"), que recibirán un resultado o un error. Las funciones de consumo pueden registrarse (suscribirse) utilizando los métodos ".then" y ".catch".

THEN

El más importante y fundamental es ".then". La sintaxis es:

```
promise.then(
  function(result) { /* manejar un resultado exitoso */ },
  function(error) { /* manejar un error */ }
);
```

El primer argumento de ".then" es una función que se ejecuta cuando se resuelve la promesa y recibe el resultado.

El segundo argumento de ".then" es una función que se ejecuta cuando se rechaza la promesa y recibe el error.

Por ejemplo, aquí hay una reacción a una promesa resuelta con éxito:

```
let promise = new Promise(function(resolve, reject) {
  setTimeout(() => resolve("hecho!"), 1000);
});

// resolve ejecuta la primera función en .then
promise.then(
  result => alert(result), // muestra "hecho!" después de 1 segundo
  error => alert(error) // no se ejecuta
);
```

La primera función fue ejecutada. Y en el caso de un rechazo, el segundo:

```
let promise = new Promise(function(resolve, reject) {
  setTimeout(() => reject(new Error("Vaya!")), 1000);
});

// reject ejecuta la segunda función en .then
promise.then(
  result => alert(result), // no se ejecuta
  error => alert(error) // muestra "Error: ¡Vaya!" después de 1 segundo
);
```

Si solo nos interesan las terminaciones exitosas, entonces podemos proporcionar solo un argumento de función para ".then":

```
let promise = new Promise(resolve => {
  setTimeout(() => resolve("hecho!"), 1000);
});

promise.then(alert); // muestra "hecho!" después de 1 segundo
```

CATCH

Si solo nos interesan los errores, entonces podemos usar null como primer argumento: ".then(null, errorHandlingFunction)". O podemos usar ".catch(errorHandlingFunction)", que es exactamente lo mismo:

```
let promise = new Promise((resolve, reject) => {
  setTimeout(() => reject(new Error("Vaya!")), 1000);
});

// .catch(f) es lo mismo que promise.then(null, f)
promise.catch(alert); // muestra "Error: ¡Vaya!" después de 1 segundo
```

La llamada ".catch(f)" es completamente equivalente a ".then(null, f)", es solo una forma abreviada.

LIMPIEZA: FINALLY

Al igual que hay una cláusula finally en un try {...} catch {...} normal, hay un finally en las promesas. La llamada ".finally(f)" es similar a ".then(f, f)" en el sentido de que f siempre se ejecuta cuando se resuelve la promesa: ya sea que se resuelva o rechace.

La idea de finally es establecer un manejador para realizar la limpieza y finalización después de que las operaciones se hubieran completado. Por ejemplo, detener indicadores de carga, cerrar conexiones que ya no son necesarias, etc.

Puedes pensarlo como el finalizador de la fiesta. No importa si la fiesta fue buena o mala ni cuántos invitados hubo, aún necesitamos (o al menos deberíamos) hacer la limpieza después. El código puede verse como esto:

```
new Promise((resolve, reject) => {
  /* hacer algo para tomar tiempo y luego llamar a resolve o reject */
})
  // se ejecuta cuando la promesa quedó establecida, no importa si con éxito o no
  .finally(() => stop loading indicator)
  // así el indicador de carga siempre es detenido antes de que sigamos adelante
  .then(result => show result, err => show error)
```

Sin embargo, note que finally(f) no es exactamente un alias dethen(f, f)`.

Hay diferencias importantes:

1. Un manejador finally no tiene argumentos. En finally no sabemos si la promesa es exitosa o no. Eso está bien, ya que usualmente nuestra tarea es realizar procedimientos de finalización “generales”.

2. Observa el ejemplo anterior: como puedes ver, el manejador de finally no tiene argumentos, y lo que sale de la promesa es manejado en el siguiente manejador.
   Resultados y errores pasan “a través” del manejador de finally. Estos pasan al siguiente manejador que se adecúe. Por ejemplo, aquí el resultado se pasa a través de finally al then que le sigue:

```
new Promise((resolve, reject) => {
  setTimeout(() => resolve("valor"), 2000)
})
  .finally(() => alert("Promesa lista")) // se dispara primero
  .then(result => alert(result)); // <-- .luego muestra "valor"
```

   Como puedes ver, el “valor” devuelto por la primera promesa es pasado a través de finally al siguiente then.
   Esto es muy conveniente, porque finally no está destinado a procesar el resultado de una promesa. Como dijimos antes, es el lugar para hacer la limpieza general sin importar cuál haya sido el resultado. Y aquí, el ejemplo de un error. Vemos cómo se pasa a través de finally a catch:

```
new Promise((resolve, reject) => {
  throw new Error("error");
})
  .finally(() => alert("Promesa lista"))  // primero dispara
  .catch(err => alert(err));  // <-- .catch muestra el error
```

1. Un manejador de finally no debe devolver nada. Y si lo hace, el valor devuelto es ignorado silenciosamente.
   La única excepción a esta regla se da cuando el manejador mismo de finally dispara un error. En ese caso, este error pasa al siguiente manejador de error en lugar del resultado previo al finally. Para summarizar:
   - Un manejador finally no obtiene lo que resultó del manejador previo (no tiene argumentos). Ese resultado es pasado a través de él al siguiente manejador.
   - Si el manejador de finally devuelve algo, será ignorado.
   - Cuando es finally el que dispara el error, la ejecución pasa al manejador de error más cercano.

Estas características son de ayuda y hacen que las cosas funcionen tal como corresponde si “finalizamos” con finally como se supone: con procedimientos de limpieza genéricos.

Ejemplo: loadScript

A continuación, veamos ejemplos más prácticos de cómo las promesas pueden ayudarnos a escribir código asincrónico. Tomemos, del capítulo anterior, la función loadScript para cargar un script. Esta es la variante basada callback, solo para recordarnos:

```
function loadScript(src, callback) {
  let script = document.createElement('script');
  script.src = src;

  script.onload = () => callback(null, script);
  script.onerror = () => callback(new Error(`Error de carga de script para $ {src}`));

  document.head.append(script);
}
```

Reescribámoslo usando Promesas.

La nueva función loadScript no requerirá una callback. En su lugar, creará y devolverá un objeto Promise que se resuelve cuando se completa la carga. El código externo puede agregar manejadores (funciones de suscripción) usando .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(`Error de carga de script para $ {src}`));

    document.head.append(script);
  });
}
```

Uso:

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

promise.then(
  script => alert(`${script.src} está cargado!`),
  error => alert(`Error: ${error.message}`)
);

promise.then(script => alert('Otro manejador...'));
```

Podemos ver inmediatamente algunos beneficios sobre el patrón basado en callback:

Promesas	Callbacks
Las promesas nos permiten hacer las cosas en el orden natural. Primero, ejecutamos `loadScript (script)`, y entonces, `.then` escribimos qué hacer con el resultado.	Debemos tener una función `callback` a nuestra disposición al llamar a 'loadScript(script, callback)'. En otras palabras, debemos saber qué hacer con el resultado *antes* de llamar a `loadScript`.
Podemos llamar a ".then" en una promesa tantas veces como queramos. Cada vez que lo hacemos estamos agregando un nuevo "fan", una nueva función de suscripción, a la "lista de suscripción". Más sobre esto en el próximo capítulo: Encadenamiento de promesas.	Solo puede haber un callback.

Encadenamiento de Promesa

Las promesas proporcionan un par de maneras para hacerlo. En este capítulo cubrimos el encadenamiento de promesas. Se ve así:

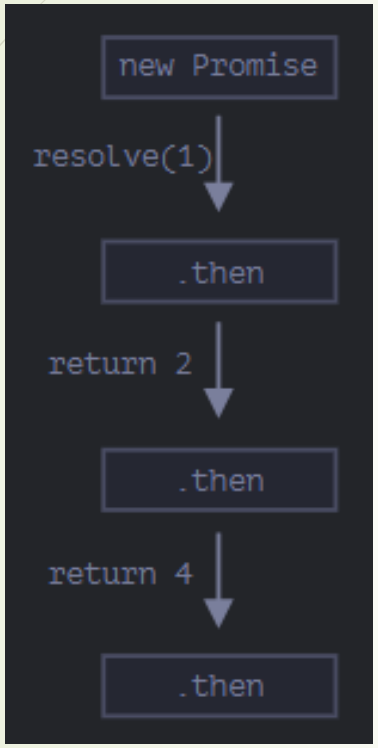
```
new Promise(function(resolve, reject) {

  setTimeout(() => resolve(1), 1000); // (*)

}).then(function(result) { // (**)

  alert(result); // 1
  return result * 2;

}).then(function(result) { // (***)

  alert(result); // 2
  return result * 2;

}).then(function(result) {

  alert(result); // 4
  return result * 2;

});
```

La idea es que el resultado pase a través de la cadena de manejadores ".then".

Aquí el flujo es:

1. La promesa inicial se resuelve en 1 segundo (*),
2. Entonces se llama el manejador .then (**), que a su vez crea una nueva promesa (resuelta con el valor 2).
3. El siguiente .then (***) obtiene el resultado del anterior, lo procesa (duplica) y lo pasa al siguiente manejador.
4. ...y así sucesivamente.

A medida que el resultado se pasa a lo largo de la cadena de controladores, podemos ver una secuencia de llamadas de alerta: 1 → 2 → 4.

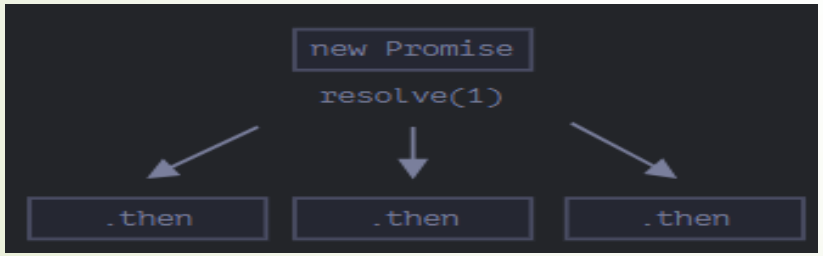


Todo funciona, porque cada llamada a promise.then devuelve una nueva promesa, para que podamos llamar al siguiente “.then” con ella. Cuando un controlador devuelve un valor, se convierte en el resultado de esa promesa, por lo que se llama al siguiente “.then”.

Un error clásico de principiante: técnicamente también podemos agregar muchos '.then' a una sola promesa: eso no es encadenamiento. Por ejemplo:

```
let promise = new Promise(function(resolve, reject) {
  setTimeout(() => resolve(1), 1000);
});

promise.then(function(result) {
  alert(result); // 1
  return result * 2;
});

promise.then(function(result) {
  alert(result); // 1
  return result * 2;
});

promise.then(function(result) {
  alert(result); // 1
  return result * 2;
});
```

Lo que hicimos aquí fue añadir varios controladores a una sola promesa. No se pasan el resultado el uno al otro; en su lugar, lo procesan de forma independiente. Aquí está la imagen (compárala con el encadenamiento anterior):



Todos los '.then' en la misma promesa obtienen el mismo resultado: el resultado de esa promesa. Entonces, en el código sobre todo alert muestra lo mismo: 1. En la práctica, rara vez necesitamos múltiples manejadores para una promesa. El encadenamiento se usa mucho más a menudo.

Devolviendo promesas

Un controlador (“handler”), utilizado en “.then(handler)”, puede crear y devolver una promesa. En ese caso, otros manejadores esperan hasta que se estabilice (resuelva o rechace) y luego obtienen su resultado. Por ejemplo:

```
new Promise(function(resolve, reject) {

  setTimeout(() => resolve(1), 1000);

}).then(function(result) {

  alert(result); // 1

  return new Promise((resolve, reject) => { // (*)
    setTimeout(() => resolve(result * 2), 1000);
  });

}).then(function(result) { // (**)

  alert(result); // 2

  return new Promise((resolve, reject) => {
    setTimeout(() => resolve(result * 2), 1000);
  });

}).then(function(result) {

  alert(result); // 4

});
```

En este código el primer “.then” muestra 1 y devuelve new Promise(...) en la línea (*). Después de un segundo, se resuelve, y el resultado (el argumento de resolve, aquí es result * 2) se pasa al controlador del segundo “.then”. Ese controlador está en la línea (**), muestra 2 y hace lo mismo.

Por lo tanto, la salida es la misma que en el ejemplo anterior: 1 → 2 → 4, pero ahora con 1 segundo de retraso entre las llamadas de alerta. Devolver las promesas nos permite construir cadenas de acciones asincrónicas.

El ejemplo: loadScript

Usemos esta función con el loadScript promisificado, definido en el capítulo anterior, para cargar los scripts uno por uno, en secuencia:

```
loadScript("/article/promise-chaining/one.js")
  .then(function(script) {
    return loadScript("/article/promise-chaining/two.js");
  })
  .then(function(script) {
    return loadScript("/article/promise-chaining/three.js");
  })
  .then(function(script) {
    // usamos las funciones declaradas en los scripts
    // para demostrar que efectivamente se cargaron
    one();
    two();
    three();
  });
```

Este código se puede acortar un poco con las funciones de flecha:

```
loadScript("/article/promise-chaining/one.js")
  .then(script => loadScript("/article/promise-chaining/two.js"))
  .then(script => loadScript("/article/promise-chaining/three.js"))
  .then(script => {
    // los scripts se cargaron, podemos usar las funciones declaradas en ellos
    one();
    two();
    three();
  });
```

Aquí cada llamada a loadScript devuelve una promesa, y el siguiente ".then" se ejecuta cuando se resuelve. Luego inicia la carga del siguiente script. Entonces los scripts se cargan uno tras otro.

Podemos agregar más acciones asincrónicas a la cadena. Tenga en cuenta que el código sigue siendo “plano”: crece hacia abajo, no a la derecha. No hay signos de la “pirámide del destino”. Técnicamente, podríamos agregar “.then” directamente a cada loadScript, así:

```
loadScript("/article/promise-chaining/one.js").then(script1 => {
  loadScript("/article/promise-chaining/two.js").then(script2 => {
    loadScript("/article/promise-chaining/three.js").then(script3 => {
      // esta función tiene acceso a las variables script1, script2 y script3
      one();
      two();
      three();
    });
  });
});
```

Este código hace lo mismo: carga 3 scripts en secuencia. Pero “crece hacia la derecha”. Entonces tenemos el mismo problema que con los callbacks. Quienes comienzan a usar promesas pueden desconocer el encadenamiento, y por ello escribirlo de esta manera. En general, se prefiere el encadenamiento.

A veces es aceptable escribir “.then” directamente, porque la función anidada tiene acceso al ámbito externo. En el ejemplo anterior, el callback más anidado tiene acceso a todas las variables script1, script2, script3. Pero eso es una excepción más que una regla.

Ejemplo más grande: fetch

En la programación “frontend”, las promesas a menudo se usan para solicitudes de red. Veamos un ejemplo extendido de esto.

Utilizaremos el método fetch para cargar la información sobre el usuario desde el servidor remoto. Tiene muchos parámetros opcionales cubiertos en capítulos separados, pero la sintaxis básica es bastante simple:

```
let promise = fetch(url);
```

Esto hace una solicitud de red a la url y devuelve una promesa. La promesa se resuelve con un objeto ‘response’ cuando el servidor remoto responde con encabezados, pero antes de que se descargue la respuesta completa.

Para leer la respuesta completa, debemos llamar al método response.text(): devuelve una promesa que se resuelve cuando se descarga el texto completo del servidor remoto, con ese texto como resultado. El siguiente código hace una solicitud a user.json y carga su texto desde el servidor:

```
fetch('/article/promise-chaining/user.json')
  // .a continuación, se ejecuta cuando el servidor remoto responde
  .then(function(response) {
    // response.text() devuelve una nueva promesa que se resuelve con el texto de res
    // cuando se carga
    return response.text();
  })
  .then(function(text) {
    // ...y aquí está el contenido del archivo remoto
    alert(text); // {"name": "iliakan", isAdmin: true}
  });
```

El objeto response devuelto por fetch también incluye el método response.json() que lee los datos remotos y los analiza como JSON. En nuestro caso, eso es aún más conveniente, así que pasemos a ello. También usaremos las funciones de flecha por brevedad:

```
// igual que el anterior, pero response.json() analiza el contenido remoto como JSON
fetch('/article/promise-chaining/user.json')
  .then(response => response.json())
  .then(user => alert(user.name)); // iliakan, tengo nombre de usuario
```

Ahora hagamos algo con el usuario cargado. Por ejemplo, podemos hacer una solicitud más a GitHub, cargar el perfil de usuario y mostrar el avatar:

```
// Hacer una solicitud para user.json
fetch('/article/promise-chaining/user.json')
  // Cárgalo como json
  .then(response => response.json())
  // Hacer una solicitud a GitHub
  .then(user => fetch(`https://api.github.com/users/${user.name}`))
  // Carga la respuesta como json
  .then(response => response.json())
  // Mostrar la imagen de avatar (githubUser.avatar_url) durante 3 segundos
  .then(githubUser => {
    let img = document.createElement('img');
    img.src = githubUser.avatar_url;
    img.className = "promise-avatar-example";
    document.body.append(img);

    setTimeout(() => img.remove(), 3000); // (*)
  });
```

El código funciona; ver comentarios sobre los detalles. Sin embargo, hay un problema potencial, un error típico para aquellos que comienzan a usar promesas.

Mire la línea (*): ¿cómo podemos hacer algo después de que el avatar haya terminado de mostrarse y se elimine? Por ejemplo, nos gustaría mostrar un formulario para editar ese usuario u otra cosa. A partir de ahora, no hay manera.

Para que la cadena sea extensible, debemos devolver una promesa que se resuelva cuando el avatar termine de mostrarse. Como esto:

```
fetch('/article/promise-chaining/user.json')
  .then(response => response.json())
  .then(user => fetch(`https://api.github.com/users/${user.name}`))
  .then(response => response.json())
  .then(githubUser => new Promise(function(resolve, reject) { // (*)
    let img = document.createElement('img');
    img.src = githubUser.avatar_url;
    img.className = "promise-avatar-example";
    document.body.append(img);

    setTimeout(() => {
      img.remove();
      resolve(githubUser); // (**)
    }, 3000);
  }))
  // se dispara después de 3 segundos
  .then(githubUser => alert(`Terminado de mostrar ${githubUser.name}`));
```

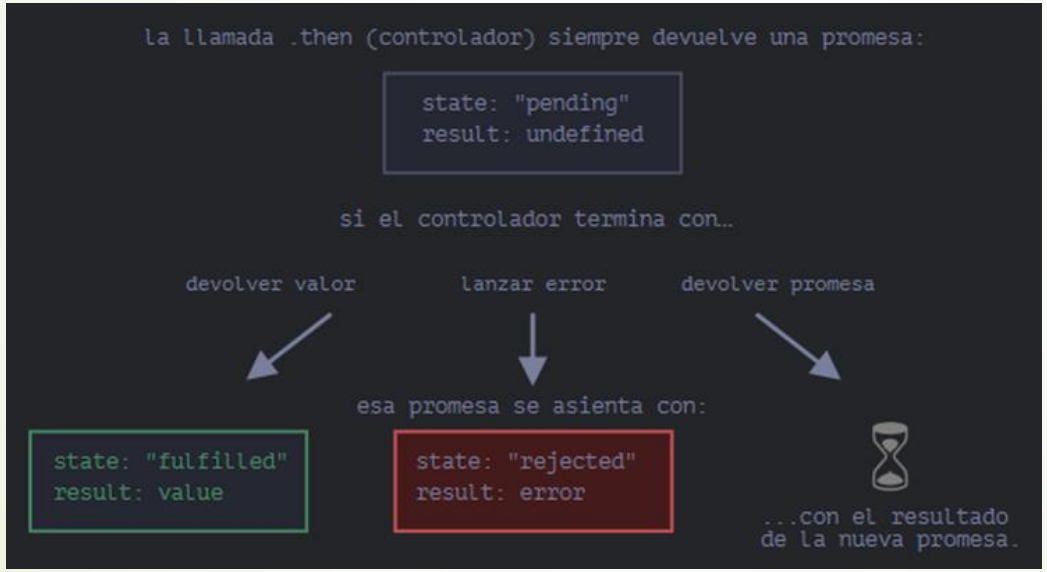
Es decir, el controlador ".then" en la línea (*) ahora devuelve new Promise, que se resuelve solo después de la llamada de resolve(githubUser) en setTimeout (**). El siguiente '.then' en la cadena esperará eso.

Como buena práctica, una acción asincrónica siempre debe devolver una promesa. Eso hace posible planificar acciones posteriores; incluso si no planeamos extender la cadena ahora, es posible que la necesitemos más adelante. Finalmente, podemos dividir el código en funciones reutilizables:

```
function loadJson(url) {
  return fetch(url)
    .then(response => response.json());
}

function loadGithubUser(name) {
  return loadJson(`https://api.github.com/users/${name}`);
}

function showAvatar(githubUser) {
  return new Promise(function(resolve, reject) {
    let img = document.createElement('img');
    img.src = githubUser.avatar_url;
    img.className = "promise-avatar-example";
    document.body.append(img);

    setTimeout(() => {
      img.remove();
      resolve(githubUser);
    }, 3000);
  });
}

// Úsalos:
loadJson('/article/promise-chaining/user.json')
  .then(user => loadGithubUser(user.name))
  .then(showAvatar)
  .then(githubUser => alert(`Finished showing ${githubUser.name}`));
  // ...
```

RESUMEN

- Si un controlador ".then" (o catch/finally, no importa) devuelve una promesa, el resto de la cadena espera hasta que ésta quede establecida (sea resuelta o rechazada). Cuando lo hace, su resultado (o error) pasa más allá. Aquí hay una imagen completa:



Manejo de errores con promesas

Las promesas encadenadas son excelentes manejando los errores. Cuando una promesa es rechazada, el control salta al manejador de rechazos más cercano. Esto es muy conveniente en la práctica.

Por ejemplo, en el código de abajo, la URL a la cual se le hace fetch es incorrecta (no existe el sitio) y al ser rechazada ".catch" maneja el error:

```
fetch('https://no-such-server.blabla') // Promesa rechazada
  .then(response => response.json())
  .catch(err => alert(err)) // TypeError: failed to fetch (El texto puede variar, dep
```

Como puedes ver, el ".catch" no tiene que escribirse inmediatamente después de la promesa. Este puede aparecer después de uno o quizás varios ".then".

O, puede ocurrir, que todo en el sitio se encuentre bien, pero la respuesta no es un JSON válido. La forma más fácil de detectar todos los errores es agregando ".catch" al final de la cadena de promesas:

```
fetch('/article/promise-chaining/user.json')
  .then(response => response.json())
  .then(user => fetch(`https://api.github.com/users/${user.name}`))
  .then(response => response.json())
  .then(githubUser => new Promise((resolve, reject) => {
    let img = document.createElement('img');
    img.src = githubUser.avatar_url;
    img.className = "promise-avatar-example";
    document.body.append(img);

    setTimeout(() => {
      img.remove();
      resolve(githubUser);
    }, 3000);
  }))
  .catch(error => alert(error.message));
```

try…catch implícito

El código de un ejecutor de promesas y de manejadores de promesas tiene embebido un "try..catch invisible". Si ocurre una excepción, esta es atrapada y es tratada como un rechazo. Por ejemplo, este código:

```
new Promise((resolve, reject) => {
  throw new Error("Whoops!");
}).catch(alert); // Error: Whoops!
```

…Hace exactamente lo mismo que este:

```
new Promise((resolve, reject) => {
  reject(new Error("Whoops!"));
}).catch(alert); // Error: Whoops!
```

El "try..catch invisible" embebido en el ejecutor detecta automáticamente el error y lo convierte en una promesa rechazada.

Esto sucede no solo en la función ejecutora, sino también en sus manejadores. Si hacemos throw dentro de una llamada a ".then", esto devolverá una promesa rechazada, por lo que el control salta al manejador de errores más cercano. Por ejemplo:

```
new Promise((resolve, reject) => {
  resolve("ok");
}).then((result) => {
  throw new Error("Whoops!"); // rechaza la promesa
}).catch(alert); // Error: Whoops!
```

Esto sucede con todos los errores, no solo los causados por la sentencia de excepción throw. Por ejemplo, un error de programación:

```
new Promise((resolve, reject) => {
  resolve("ok");
}).then((result) => {
  blabla(); // Función inexistente
}).catch(alert); // ReferenceError: blabla is not defined
```

El ".catch" del final no solo detecta rechazos explícitos, sino también los errores accidentales en los manejadores anteriores.

Rethrowing (relanzamiento de errores)

Como ya vimos, el ".catch" del final es similar a "try..catch". Podemos tener tantos manejadores ".then" como queramos, y luego usar un solo .catch al final para manejar los errores en todos ellos. En un "try..catch" normal, podemos analizar el error y quizá volver a lanzarlo si no se puede manejar. Lo mismo podemos hacer con las promesas.

Si hacemos throw dentro de ".catch", el control pasa a otro manejador de errores más cercano. Y, si manejamos el error y terminamos de forma correcta, entonces se continúa con el siguiente manejador ".then" exitoso. En el ejemplo de abajo, el ".catch" maneja el error de forma exitosa:

```
1  // Ejecución: catch -> then
2  new Promise((resolve, reject) => {
3
4    throw new Error("Whoops!");
5
6  }).catch(function(error) {
7
8    alert("Error manejado, se continuará con la ejecución del código");
9
10 }).then(() => alert("El siguiente manejador exitoso se ejecuta"));
```

Aquí el ".catch" termina de forma correcta. Entonces se ejecuta el siguiente manejador exitoso ".then".

En el siguiente ejemplo podemos ver otra situación con ".catch". El manejador (*) detecta el error y simplemente no puede manejarlo (en el ejemplo solo sabe qué hacer con un URIError), por lo que lo lanza nuevamente:

```
// Ejecución: catch -> catch
new Promise((resolve, reject) => {

  throw new Error("Whoops!");

}).catch(function(error) { // (*)

  if (error instanceof URIError) {
    // Aquí se manejaría el error
  } else {
    alert("No puedo manejar este error");

    throw error; // Lanza este error u otro error que salte en el siguiente catch.
  }

}).then(function() {
  /* Esto no se ejecuta */
}).catch(error => { // (**)

  alert(`Ocurrió un error desconocido: ${error}`);
  // No se devuelve nada => La ejecución continúa de forma normal

});
```

La ejecución salta del primer ".catch" (*) al siguiente (**) en la cadena.

Rechazos no manejados

¿Qué sucede cuanto un error no es manejado? Por ejemplo, si olvidamos agregar “.catch” al final de una cadena de promesas, como aquí:

```
1  new Promise(function() {
2    noSuchFunction(); // Aquí hay un error (no existe la función)
3  })
4    .then(() => {
5      // manejador de una o más promesas exitosas
6    }); // sin .catch al final!
```

En caso de que se genere un error, la promesa se rechaza y la ejecución salta al manejador de rechazos más cercano. Pero aquí no hay ninguno. Entonces el error se “atasca”, ya que no hay código para manejarlo. En la práctica, al igual que con los errores comunes no manejados en el código, esto significa que algo ha salido terriblemente mal.

¿Qué sucede cuando ocurre un error regular y no es detectado por try..catch? El script muere con un mensaje en la consola. Algo similar sucede con los rechazos de promesas no manejadas.

En este caso, el motor de JavaScript rastrea el rechazo y lo envía al ámbito global. Puedes ver en consola el error generado si ejecutas el ejemplo anterior. En el navegador podemos detectar tales errores usando el evento unhandledrejection:

```
window.addEventListener('unhandledrejection', function(event) {
  // el objeto event tiene dos propiedades especiales:
  alert(event.promise); // [objeto Promesa] - La promesa que fue rechazada
  alert(event.reason); // Error: Whoops! - Motivo por el cual se rechaza la promesa
});

new Promise(function() {
  throw new Error("Whoops!");
}); // No hay un .catch final para manejar el error
```

Si se produce un error, y no hay un “.catch”, se dispara unhandledrejection, y se obtiene el objeto event el cual contiene información sobre el error, por lo que podemos hacer algo con el error (manejar el error).

Usualmente estos errores no son recuperables, por lo que la mejor salida es informar al usuario sobre el problema y probablemente reportar el incidente al servidor. En entornos fuera del navegador como Node.js existen otras formas de rastrear errores no manejados.

RESUMEN

- “.catch” maneja errores de todo tipo: ya sea una llamada a reject(), o un error que arroja un manejador.
- “.then” también atrapa los errores de la misma manera si se le da el segundo argumento (que es el manejador de error).
- Debemos colocar “.catch” exactamente en los lugares donde queremos manejar los errores y saber cómo manejarlos. El manejador debe analizar los errores (los errores personalizados ayudan), y relanzar los errores desconocidos (tal vez sean errores de programación).
- Es correcto no usar “.catch” en absoluto si no hay forma de recuperarse de un error.
- En cualquier caso, deberíamos tener el evento unhandledrejection (para navegadores, o el equivalente en otros entornos) para monitorear errores no manejados e informar al usuario (y probablemente al servidor) para que nuestra aplicación nunca “simplemente muera”.

PROMISE API

- Hay 6 métodos estáticos en la clase Promise. Veremos sus casos de uso aquí.

1. Promise.all

Digamos que queremos que muchas promesas se ejecuten en paralelo y esperar hasta que todas ellas estén listas.

Por ejemplo, descargar varias URLs en paralelo y procesar su contenido en cuanto todas ellas finalicen. Para ello es Promise.all. La sintaxis es:

```
let promise = Promise.all(iterable);
```

Promise.all toma un iterable (usualmente un array de promesas) y devuelve una nueva promesa.

Esta nueva promesa es resuelta en cuanto todas las promesas listadas se resuelven, y el array de aquellos resultados se vuelve su resultado. Por ejemplo, el Promise.all debajo se resuelve después de 3 segundos, y su resultado es un array [1, 2, 3]:

```
Promise.all([
  new Promise(resolve => setTimeout(() => resolve(1), 3000)), // 1
  new Promise(resolve => setTimeout(() => resolve(2), 2000)), // 2
  new Promise(resolve => setTimeout(() => resolve(3), 1000))  // 3
]).then(alert); // 1,2,3 cuando las promesas están listas: cada promesa constituye
```

Ten en cuenta que el orden de los miembros del array es el mismo que el de las promesas que los originan. Aunque la primera promesa es la que toma más tiempo en resolverse, es aún la primera en el array de resultados.

Un truco común es mapear un array de datos de trabajo dentro de un array de promesas, y entonces envolverlos dentro de un Promise.all. Por ejemplo, si tenemos un array de URLs, podemos usar fetch en todos ellos así:

```
let urls = [
  'https://api.github.com/users/iliakan',
  'https://api.github.com/users/remy',
  'https://api.github.com/users/jeresig'
];

// "mapear" cada url a la promesa de su fetch
let requests = urls.map(url => fetch(url));

// Promise.all espera hasta que todas la tareas estén resueltas
Promise.all(requests)
  .then(responses => responses.forEach(
    response => alert(`${response.url}: ${response.status}`)
  ));
```

Un mayor ejemplo con fetch: la búsqueda de información de usuario para un array de usuarios de GitHub por sus nombres (o podríamos buscar un array de bienes por sus “id”, la lógica es idéntica):

```
let names = ['iliakan', 'remy', 'jeresig'];

let requests = names.map(name => fetch(`https://api.github.com/users/${name}`));

Promise.all(requests)
  .then(responses => {
    // todas las respuestas son resueltas satisfactoriamente
    for(let response of responses) {
      alert(`${response.url}: ${response.status}`); // muestra 200 por cada url
    }

    return responses;
  })
  // mapea el array de resultados dentro de un array de response.json() para leer
  .then(responses => Promise.all(responses.map(r => r.json())))
  // todas las respuestas JSON son analizadas: "users" es el array de ellas
  .then(users => users.forEach(user => alert(user.name)));
```

Si cualquiera de las promesas es rechazada, la promesa devuelta por Promise.all inmediatamente rechaza: “reject” con ese error. Por ejemplo:

```
Promise.all([
  new Promise((resolve, reject) => setTimeout(() => resolve(1), 1000)),
  new Promise((resolve, reject) => setTimeout(() => reject(new Error("Whoops!")), 2000)),
  new Promise((resolve, reject) => setTimeout(() => resolve(3), 3000))
]).catch(alert); // Error: Whoops!
```

Aquí la segunda promesa se rechaza en dos segundos. Esto lleva a un rechazo inmediato de Promise.all, entonces “.catch” se ejecuta: el error del rechazo se vuelve la salida del Promise.all entero.

2. Promise.allSettled

Promise.all rechaza como un todo si cualquiera de sus promesas es rechazada. Esto es bueno para los casos de “todo o nada”, cuando necesitamos que todos los resultados sean exitosos para proceder:

```
Promise.all([
  fetch('/template.html'),
  fetch('/style.css'),
  fetch('/data.json')
]).then(render); // el método render necesita los resultados de todos los fetch
```

Promise.allSettled solo espera que todas las promesas se resuelvan sin importar sus resultados. El array resultante tiene:

- {status:"fulfilled", value:result} para respuestas exitosas,
- {status:"rejected", reason:error} para errores.

Por ejemplo, quisiéramos hacer “fetch” de la información de múltiples usuarios. Incluso si uno falla, aún estaremos interesados en los otros. Usemos Promise.allSettled:

```
let urls = [
  'https://api.github.com/users/iliakan',
  'https://api.github.com/users/remy',
  'https://no-such-url'
];

Promise.allSettled(urls.map(url => fetch(url)))
  .then(results => { // (*)
    results.forEach((result, num) => {
      if (result.status == "fulfilled") {
        alert(`${urls[num]}: ${result.value.status}`);
      }
      if (result.status == "rejected") {
        alert(`${urls[num]}: ${result.reason}`);
      }
    });
  });
```

El results de la línea (*) de arriba será:

```
[
  {status: 'fulfilled', value: ...response...},
  {status: 'fulfilled', value: ...response...},
  {status: 'rejected', reason: ...error object...}
]
```

Entonces para cada promesa obtendremos su estado y value/error.

3. Polyfill

Si el browser no soporta Promise.allSettled, es fácil implementarlo:

```
if (!Promise.allSettled) {
 const rejectHandler = reason => ({ status: 'rejected', reason });
 const resolveHandler = value => ({ status: 'fulfilled', value });
 Promise.allSettled = function (promises) {
  const convertedPromises = promises.map(p => Promise.resolve(p).then(resolveHandler, rejectHandler));
  return Promise.all(convertedPromises);
 };
}
```

En este código, promises.map toma los valores de entrada, los transforma en promesas (por si no lo eran) con p => Promise.resolve(p), entonces agrega un manejador ".then" a cada una.

Este manejador ("handler") transforma un resultado exitoso value en {status:'fulfilled', value}, y un error reason en {status:'rejected', reason}. Ese es exactamente el formato de Promise.allSettled. Ahora podemos usar Promise.allSettled para obtener el resultado de todas las promesas dadas incluso si algunas son rechazadas.

4. Promise.race

Similar a Promise.all, pero espera solamente por la primera respuesta y obtiene su resultado (o error). Su sintaxis es:

```
1  let promise = Promise.race(iterable);
```

Por ejemplo, aquí el resultado será 1:

```
Promise.race([
  new Promise((resolve, reject) => setTimeout(() => resolve(1), 1000)),
  new Promise((resolve, reject) => setTimeout(() => reject(new Error("Whoops!")), 2000)),
  new Promise((resolve, reject) => setTimeout(() => resolve(3), 3000))
]).then(alert); // 1
```

La primera promesa fue la más rápida, por lo que se vuelve resultado. En cuanto una promesa responde, “gana la carrera”, y todos los resultados o errores posteriores son ignorados.

5. Promise.any

Es similar a Promise.race, pero espera solamente por la primera promesa cumplida y obtiene su resultado. Si todas las promesas fueron rechazadas, entonces la promesa que devuelve es rechazada con AggregateError, un error especial que almacena los errores de todas las promesas en su propiedad errors. La sintaxis es:

```
let promise = Promise.any(iterable);
```

Por ejemplo, aquí el resultado será 1:

```
Promise.any([
  new Promise((resolve, reject) => setTimeout(() => reject(new Error("Whoops!")), 1000)),
  new Promise((resolve, reject) => setTimeout(() => resolve(1), 2000)),
  new Promise((resolve, reject) => setTimeout(() => resolve(3), 3000))
]).then(alert); // 1
```

La primera promesa fue la más rápida, pero fue rechazada entonces devuelve el resultado de la segunda. Una vez que la primera promesa cumplida “gana la carrera”, los demás resultados serán ignorados. Aquí hay un ejemplo donde todas las promesas fallan:

```
Promise.any([
  new Promise((resolve, reject) => setTimeout(() => reject(new Error("Ouch!")), 1000)),
  new Promise((resolve, reject) => setTimeout(() => reject(new Error("Error!")), 2000))
]).catch(error => {
  console.log(error.constructor.name); // AggregateError
  console.log(error.errors[0]); // Error: Ouch!
  console.log(error.errors[1]); // Error: Error!
});
```

Como puedes ver, los objetos de error de las promesas que fallaron están disponibles en la propiedad errors del objeto AggregateError.

6. Promise.resolve/reject

Los métodos Promise.resolve y Promise.reject son raramente necesitados en código moderno porque la sintaxis async/await (que veremos luego) las hace algo obsoletas.

Las tratamos aquí para completar la cobertura y por aquellos casos que por algún motivo no puedan usar async/await.

Promise.resolve

Promise.resolve(value) crea una promesa resuelta con el resultado value. Tal como:

```
let promise = new Promise(resolve => resolve(value));
```

El método es usado por compatibilidad, cuando se espera que una función devuelva una promesa.

Por ejemplo, la función loadCached abajo busca una URL y recuerda (en caché) su contenido. Futuros llamados con la misma URL devolverá el contenido de caché, pero usa Promise.resolve para hacer una promesa de él y así el valor devuelto es siempre una promesa:

```
let cache = new Map();

function loadCached(url) {
  if (cache.has(url)) {
    return Promise.resolve(cache.get(url)); // (*)
  }

  return fetch(url)
    .then(response => response.text())
    .then(text => {
      cache.set(url,text);
      return text;
    });
}
```

Podemos escribir loadCached(url).then(...) porque se garantiza que la función devuelve una promesa. Siempre podremos usar ".then" después de loadCached. Ese es el propósito de Promise.resolve en la línea (*).

Promise.reject

Promise.reject(error) crea una promesa rechazada con error. Tal como:

```
1  let promise = new Promise((resolve, reject) => reject(error));
```

En la práctica este método casi nunca es usado.

RESUMEN

- Existen 6 métodos estáticos de la clase Promise:

1. Promise.all(promises) – espera que todas las promesas se resuelvan y devuelve un array de sus resultados. Si cualquiera es rechazada se vuelve el error de Promise.all y los demás resultados son ignorados.
2. Promise.allSettled(promises) (método recientemente añadido) – espera que toda las promesas respondan y devuelve sus resultados como un array de objetos con:
   - status: "fulfilled" o "rejected"
   - value (si fulfilled) o reason (si rejected).
3. Promise.race(promises) – espera a la primera promesa que responda y aquel resultado o error se vuelve su resultado o error.
4. Promise.any(promises) (método recientemente añadido) – espera por la primera promesa que se cumpla y devuelve su resultado. Si todas las promesas son rechazadas, AggregateError se vuelve el error de Promise.any.
5. Promise.resolve(value) – crea una promesa resuelta con el "value" dado.
6. Promise.reject(error) – crea una promesa rechazada con el "error" dado.

- Promise.all es probablemente el más común en la práctica.

Promisificación

“Promisificación” es una simple transformación. Es la conversión de una función que acepta un callback a una función que devuelve una promesa.

A menudo estas transformaciones son necesarias en la vida real ya que muchas funciones y librerías están basadas en callbacks, pero las promesas son más convenientes así que tiene sentido promisificarlas. Veamos un ejemplo. Aquí tenemos loadScript(src, callback) del artículo Introducción: callbacks.

```
function loadScript(src, callback) {
  let script = document.createElement('script');
  script.src = src;

  script.onload = () => callback(null, script);
  script.onerror = () => callback(new Error(`Error de carga de script ${src}`));

  document.head.append(script);
}

// uso:
// loadScript('path/script.js', (err, script) => {...})
```

La función carga un script con el src dado, y llama a callback(err) en caso de error o callback(null, script) en caso de carga exitosa. Esto está ampliamente acordado en el uso de callbacks, lo hemos visto antes. Vamos a promisificarla.

Haremos una función nueva loadScriptPromise(src) que va a hacer lo mismo (carga el script), pero devuelve una promesa en vez de usar callbacks. Es decir: pasamos solamente src (sin callback) y obtenemos una promesa de vuelta, que resuelve con script cuando la carga fue exitosa y rechaza con error en caso contrario. Aquí está:

```
let loadScriptPromise = function(src) {
  return new Promise((resolve, reject) => {
    loadScript(src, (err, script) => {
      if (err) reject(err);
      else resolve(script);
    });
  });
};

// uso:
// loadScriptPromise('path/script.js').then(...)
```

Como podemos ver, la nueva función es un “wrapper” (una función contenedora) que envuelve la función loadScript original. La llama proveyendo su propio callback y la traduce a una promesa resolve/reject.

Ahora loadScriptPromise se adapta bien a un código basado en promesas. Si nos gustan más las promesas que los callbacks (y pronto veremos más motivos para ello), la usaremos en su lugar. En la práctica podemos necesitar promisificar más de una función, así que tiene sentido usar un ayudante. Lo llamamos promisify(f): esta acepta la función a promisificar f y devuelve una función contenedora (wrapper).

```
function promisify(f) {
  return function (...args) { // devuelve una función contenedora (*)
    return new Promise((resolve, reject) => {
      function callback(err, result) { // nuestro callback personalizado para f (**)
        if (err) {
          reject(err);
        } else {
          resolve(result);
        }
      }

      args.push(callback); // adjunta nuestro callback personalizado al final de los

      f.call(this, ...args); // llama a la función original
    });
  };
}

// uso:
let loadScriptPromise = promisify(loadScript);
loadScriptPromise(...).then(...);
```

El código puede verse complicado, pero es esencialmente lo mismo que escribimos arriba al promisificar la función loadScript.

Una llamada a promisify(f) devuelve una función contenedora que envuelve a f (*). Este contenedor devuelve una promesa y redirige el llamado a la f original, siguiendo el resultado en el callback personalizado (**). Aquí promisify asume que la función original espera un callback con dos argumentos (err, result). Eso es lo que usualmente encontramos. Entonces nuestro callback personalizado está exactamente en el formato correcto, y promisify funciona muy bien para tal caso.

¿Y si la f original espera un callback con más argumentos callback(err, res1, res2)? Podemos mejorar el ayudante. Hagamos una versión de promisify más avanzada.

- Cuando la llamamos como promisify(f), debe funcionar igual que en la versión previa.
- Cuando la llamamos como promisify(f, true), debe devolver una promesa que resuelve con el array de resultados del callback. Esto es para callbacks con muchos argumentos.

```
// promisify(f, true) para conseguir array de resultados
function promisify(f, manyArgs = false) {
  return function (...args) {
    return new Promise((resolve, reject) => {
      function callback(err, ...results) { // Nuestro callback personalizado para f
        if (err) {
          reject(err);
        } else {
          // Devolver todos los resultados del callback si "manyArgs" es especificado
          resolve(manyArgs ? results : results[0]);
        }
      }

      args.push(callback);

      f.call(this, ...args);
    });
  };
}

// Uso:
f = promisify(f, true);
f(...).then(arrayOfResults => ..., err => ...);
```

Como puedes ver es esencialmente lo mismo de antes, pero resolve es llamado con solo uno o con todos los argumentos dependiendo del valor de manyArgs. Para formatos más exóticos de callback, como aquellos sin err en absoluto: callback(result), podemos promisificarlos manualmente sin usar el ayudante.

Async / Await

Existe una sintaxis especial para trabajar con promesas de una forma más confortable, llamada "async/await". Es sorprendentemente fácil de entender y usar.

Funciones async

Comencemos con la palabra clave async. Puede ser ubicada delante de una función como aquí:

```
async function f() {
  return 1;
}
```

La palabra "async" ante una función significa solamente una cosa: que la función siempre devolverá una promesa. Otros valores serán envueltos y resueltos en una promesa automáticamente. Por ejemplo, esta función devuelve una promesa resuelta con el resultado de 1; Probémosla:

```
async function f() {
  return 1;
}

f().then(alert); // 1
```

...Podríamos explícitamente devolver una promesa, lo cual sería lo mismo:

```
async function f() {
  return Promise.resolve(1);
}

f().then(alert); // 1
```

Entonces, async se asegura de que la función devuelva una promesa, o envuelve las no promesas y las transforma en una. Bastante simple, ¿correcto? Pero no solo eso. Hay otra palabra, await, que solo trabaja dentro de funciones async y es muy interesante.

Await

La sintaxis:

```
// funciona solamente dentro de funciones async
let value = await promise;
```

await hace que JavaScript espere hasta que la promesa responda y devuelve su resultado. Aquí hay un ejemplo con una promesa que resuelve en 1 segundo:

```
async function f() {

  let promise = new Promise((resolve, reject) => {
    setTimeout(() => resolve("¡Hecho!"), 1000)
  });

  let result = await promise; // espera hasta que la promesa se resuelva (*)

  alert(result); // "¡Hecho!"
}

f();
```

La ejecución de la función es pausada en la línea (*) y se reanuda cuando la promesa responde, con result volviéndose su resultado. Entonces el código arriba muestra “¡Hecho!” en un segundo.

Enfaticemos: **await** literalmente suspende la ejecución de la función hasta que se establezca la promesa, y luego la reanuda con el resultado de la promesa. Eso no cuesta ningún recurso de CPU, porque el motor de JavaScript puede hacer otros trabajos mientras tanto: ejecutar otros scripts, manejar eventos, etc. Es simplemente una sintaxis más elegante para tener el resultado de una promesa que “promise.then”, es más fácil de leer y de escribir.

Nota: “await” solo funciona dentro de una función async.

Tomemos el ejemplo showAvatar() del capítulo Encadenamiento de promesas y rescribámoslo usando async/await:

1. Necesitaremos reemplazar los llamados ".then" con await.
2. También debemos hacer que la función sea async para que aquellos funcionen.

```
async function showAvatar() {

  // leer nuestro JSON
  let response = await fetch('/article/promise-chaining/user.json');
  let user = await response.json();

  // leer usuario github
  let githubResponse = await fetch(`https://api.github.com/users/${user.name}`);
  let githubUser = await githubResponse.json();

  // muestra el avatar
  let img = document.createElement('img');
  img.src = githubUser.avatar_url;
  img.className = "promise-avatar-example";
  document.body.append(img);

  // espera 3 segundos
  await new Promise((resolve, reject) => setTimeout(resolve, 3000));

  img.remove();

  return githubUser;
}

showAvatar();
```

Bien limpio y fácil de leer, ¿no es cierto? Mucho mejor que antes.

Métodos de clase Async: Para declarar un método de clase async, simplemente se le antepone async:

```
class Waiter {
  async wait() {
    return await Promise.resolve(1);
  }
}

new Waiter()
  .wait()
  .then(alert); // 1 (lo mismo que (result => alert(result)))
```

Manejo de Error

Si una promesa se resuelve normalmente, entonces await promise devuelve el resultado. Pero en caso de rechazo, dispara un error, tal como si hubiera una instrucción throw en aquella línea. Este código:

```
async function f() {
  await Promise.reject(new Error("Whoops!"));
}
```

...es lo mismo que esto:

```
async function f() {
  throw new Error("Whoops!");
}
```

En situaciones reales, la promesa tomará algún tiempo antes del rechazo. En tal caso habrá un retardo antes de que await dispare un error. Podemos atrapar tal error usando try..catch, de la misma manera que con un throw normal:

```
async function f() {

  try {
    let response = await fetch('http://no-such-url');
  } catch(err) {
    alert(err); // TypeError: failed to fetch
  }
}

f();
```

En el caso de un error, el control salta al bloque catch. Podemos también envolver múltiples líneas:

```
async function f() {

  try {
    let response = await fetch('/no-user-here');
    let user = await response.json();
  } catch(err) {
    // atrapa errores tanto en fetch como en response.json
    alert(err);
  }
}

f();
```

Si no tenemos try..catch, entonces la promesa generada por el llamado de la función async f() se vuelve rechazada. Podemos añadir ".catch" para manejarlo:

```
async function f() {
  let response = await fetch('http://no-such-url');
}

// f() se vuelve una promesa rechazada
f().catch(alert); // TypeError: failed to fetch // (*)
```

Si olvidamos añadir ".catch" allí, obtendremos un error de promesa no manejado (visible en consola). Podemos atrapar tales errores usando un manejador de evento global unhandledrejection.

RESUMEN

- El comando async antes de una función tiene dos efectos:
  1. Hace que siempre devuelva una promesa.
  2. Permite que sea usado await dentro de ella.
- El comando await antes de una promesa hace que JavaScript espere hasta que la promesa responda. Entonces:
  1. Si es un error, la excepción es generada — lo mismo que si throw error fuera llamado en ese mismo lugar.
  2. De otro modo, devuelve el resultado.
- Juntos proveen un excelente marco para escribir código asincrónico que es fácil de leer y escribir.
- Con async/await raramente necesitamos escribir promise.then/catch, pero aún no deberíamos olvidar que están basados en promesas porque a veces (ej. como en el nivel superior de código) tenemos que usar esos métodos. También Promise.all es adecuado cuando esperamos por varias tareas simultáneas.