Free email newsletter: "ES.next News"

2014-10-05

ECMAScript 6 promises (2/2): the API

Labels: async, dev, esnext, javascript, jslang, promises

This blog post is outdated. Please read chapter "Promises for asynchronous programming" in "Exploring ES6".

This blog post is an introduction to asynchronous programming via promises in general and the ECMAScript 6 (ES6) promise API in particular. It is second in a series of two posts – part one explains foundations of asynchronous programming (which you may need to learn in order to fully understand this post).

Given that the ECMAScript 6 promise API is easy to polyfill for ECMAScript 5, I'm mainly using function expressions and not ECMAScript 6 arrow functions, even though the latter are much less verbose.

1. Promises

Promises are a pattern that helps with one particular kind of asynchronous programming: functions (or methods) that return their results asynchronously. To implement such a function, you return a *promise*, an object that is a placeholder for the result. The caller of the function registers callbacks with the promise to be notified once the result has been computed. The function sends the result via the promise.

The de-facto standard for JavaScript promises is called Promises/A+ [1]. The ECMAScript 6 promise API follows that standard.

2. A first example

Let's look at a first example, to give you a taste of what working with promises is like.

With Node.js-style callbacks, reading a file asynchronously looks like this:

```
fs.readFile('config.json',
    function (error, text) {
        if (error) {
            console.error('Error while reading config file');
        } else {
            try {
                var obj = JSON.parse(text);
                console.log(JSON.stringify(obj, null, 4));
            } catch (e) {
                console.error('Invalid JSON in file');
            }
        }
    });
```

With promises, the same functionality is implemented like this:

```
readFilePromisified('config.json')
.then(function (text) { // (A)
    var obj = JSON.parse(text);
    console.log(JSON.stringify(obj, null, 4));
})
```

Dr. Axel Rauschmayer

```
    .catch(function (reason) { // (B)
        // File read error or JSON SyntaxError
        console.error('An error occurred', reason);
    });
```

There are still callbacks, but they are provided via methods that are invoked on the result (`then()` and `catch()`). The error callback in line (B) is convenient in two ways: First, it's a single style of handling errors. Second, you can handle the errors of both `readFilePromisified()` and the callback from line (A).

3. Creating and using promises

Let's look at how promises are operated from the producer and the consumer side.

3.1. Producing a promise

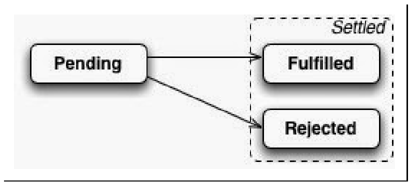
As a producer, you create a promise and send a result via it:

```
    var promise = new Promise(
        function (resolve, reject) { // (A)
            ...
            if (...) {
                resolve(value); // success
            } else {
                reject(reason); // failure
            }
        });
```

A promise is always in either one of three (mutually exclusive) states:

- Pending: the result hasn't been computed, yet
- Fulfilled: the result was computed successfully
- Rejected: a failure occurred during computation

A promise is *settled* (the computation it represents has finished) if it is either fulfilled or rejected. A promise can only be settled once and then stays settled. Subsequent attempts to settle it have no effect.



The parameter of `new Promise()` (starting in line (A)) is called an *executor*:

- If the computation went well, the executor sends the result via `resolve()`. That usually fulfills the promise (it may not, if you resolve with a promise, as explained later).
- If an error happened, the executor notifies the promise consumer via `reject()`. That always rejects the promise.

3.2. Consuming a promise

As a consumer of `promise`, you are notified of a fulfillment or a rejection via *reactions* – callbacks that you register with the method `then()`:

```
    promise.then(
        function (value) { /* fulfillment */ },
        function (reason) { /* rejection */ }
    );
```

What makes promises so useful for asynchronous functions (with one-off results) is that once a promise is settled, it doesn't change anymore. Furthermore, there are never any race conditions, because it doesn't matter whether you invoke `then()` before or after a promise is settled:

- In the former case, the appropriate reaction is called as soon as the promise is settled.

Labels

dev (592)
javascript (400)
computers (316)
life (194)
jslang (179)
esnext (156)
apple (107)
webdev (95)
mobile (83)
scitech (50)
hack (49)
mac (47)
google (39)
java (37)
ios (33)
business (32)
video (32)
clientjs (31)
hci (27)
entertainment (26)
nodejs (26)

society (26)
browser (25)
firefox (25)
html5 (24)
ipad (24)
movie (23)
psychology (22)
2ality (18)
tv (18)
android (17)
social (17)
chrome (16)
fun (16)
jsmodules (16)
tablet (16)
humor (15)
politics (15)
web (15)
cloud (14)
hardware (14)
microsoft (14)
software engineering (13)
blogging (12)
gaming (12)
eclipse (11)
gwt (11)
numbers (11)
programming languages (11)
app store (10)
media (10)
nature (10)
security (10)
semantic web (10)
software (10)
twitter (10)
webos (10)
12quirks (9)
education (9)
jstools (9)
photo (9)
webcomponents (9)
windows 8 (9)
async (8)
idea (8)
iphone (8)
itunes (8)
scifi-fantasy (8)
app (7)
babel (7)
bookmarklet (7)
chromeos (7)
english (7)
es proposal (7)

- In the latter case, the promise result (fulfillment value or rejection value) is cached and handed to the appropriate reaction “immediately” (queued as a task).

3.3. Only handling fulfillments or rejections

If you are only interested in fulfillments, you can omit the second parameter of `then()`:

```
promise.then(
    function (value) { /* fulfillment */ }
);
```

If you are only interested in rejections, you can omit the first parameter. The method `catch()` is a more compact way of doing the same thing.

```
promise.then(
    null,
    function (reason) { /* rejection */ }
);

// Equivalent:
promise.catch(
    function (reason) { /* rejection */ }
);
```

It is recommended to use `then()` exclusively for fulfillments and `catch()` for errors, because it nicely labels callbacks and because you can handle the rejections of multiple promises at the same time (how is explained later).

4. Examples

Let’s use these basic building blocks in a few examples.

4.1. Example: promisifying XMLHttpRequest

The following is a promise-based function that performs an HTTP GET via the event-based XMLHttpRequest API:

```
function httpGet(url) {
    return new Promise(
        function (resolve, reject) {
            var request = new XMLHttpRequest();
            request.onreadystatechange = function () {
                if (this.status === 200) {
                    // Success
                    resolve(this.response);
                } else {
                    // Something went wrong (404 etc.)
                    reject(new Error(this.statusText));
                }
            }
            request.onerror = function () {
                reject(new Error(
                    'XMLHttpRequest Error: '+this.statusText));
            };
            request.open('GET', url);
            request.send();
        });
}
```

This is how you use `httpGet()`:

```
httpGet('http://example.com/file.txt')
.then(
    function (value) {
        console.log('Contents: ' + value);
    },
    function (reason) {
        console.error('Something went wrong', reason);
    });
```

fringe (7)
html (7)
jsint (7)
jsshell (7)
thunderbolt (7)
webapp (7)
advancedjs (6)
blogger (6)
crowdsourcing (6)
latex (6)
lion (6)
promises (6)
ted (6)
book (5)
environment (5)
gadget (5)
googleio (5)
intel (5)
jsarrays (5)
jshistory (5)
layout (5)
light peak (5)
michael j. fox (5)
music (5)
pdf (5)
polymer (5)
shell (5)
tc39 (5)
template literals (5)
underscorejs (5)
vlc (5)
__proto__ (4)
coffeescript (4)
concurrency (4)
dart (4)
facebook (4)
gimp (4)
googleplus (4)
health (4)
howto (4)
hp (4)
javafx (4)
kindle (4)
leopard (4)
macbook (4)
motorola (4)
münchen (4)
occupy (4)
pl fundamentals (4)
presenting (4)
publishing (4)
series (4)
textbook (4)
web design (4)

4.2. Example: delaying an activity

Let's implement `setTimeout()` as the promise-based function `delay()` (similar to `Q.delay()`).

```
function delay(ms) {
    return new Promise(function (resolve, reject) {
        setTimeout(resolve, ms); // (A)
    });
}

// Using delay():
delay(5000).then(function () { // (B)
    console.log('5 seconds have passed!')
});
```

Note that in line (A), we are calling `resolve` with zero parameters, which is the same as calling `resolve(undefined)`. We don't need the fulfillment value in line (B), either and simply ignore it. Just being notified is enough here.

4.3. Example: timing out a promise

```
function timeout(ms, promise) {
    return new Promise(function (resolve, reject) {
        promise.then(resolve);
        setTimeout(function () {
            reject(new Error('Timeout after '+ms+' ms')); // (A)
        }, ms);
    });
}
```

Note that the rejection after the timeout (in line (A)) does not cancel the request, but it does prevent the promise being fulfilled with its result.

Using `timeout()` looks like this:

```
timeout(5000, httpGet('http://example.com/file.txt'))
.then(function (value) {
    console.log('Contents: ' + value);
})
.catch(function (reason) {
    console.error('Error or timeout', reason);
});
```

5. Chaining `then()`

The result of a method call

```
P.then(onFulfilled, onRejected)
```

is a new promise Q. That means that you can keep the promised-based control flow going by invoking `then()` on Q:

- Q is resolved with what is returned by either `onFulfilled` or `onRejected`.
- Q is rejected if either `onFulfilled` or `onRejected` throw an exception.

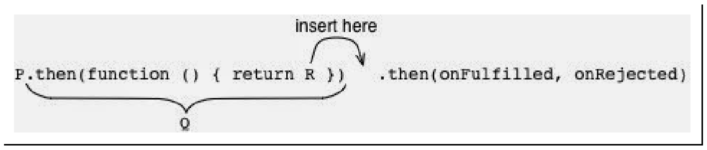
5.1. Resolving with normal values

If you resolve the promise Q returned by `then()` with a normal value, you can pick up that value via a subsequent `then()`:

```
asyncFunc()
.then(function (value1) {
    return 123;
})
.then(function (value2) {
    console.log(value2); // 123
});
```

5.2. Resolving with thenables

You can also resolve the promise Q returned by `then()` with a *thenable* R. A thenable is any object that has a promise-style method `then()`. Thus, promises are thenable. Resolving with R (e.g. by returning it from `onFulfilled`) means that it is inserted “after” Q: R’s settlement is forwarded to Q’s `onFulfilled` and `onRejected` callbacks. In a way, Q becomes R.


insert here
P.then(function () { return R })
.then(onFulfilled, onRejected)
Q

The main use for this mechanism is to flatten nested `then()` calls, like in the following example:

```
asyncFunc1()
.then(function (value1) {
    asyncFunc2()
    .then(function (value2) {
        ...
    });
})
```

The flat version looks like this:

```
asyncFunc1()
.then(function (value1) {
    return asyncFunc2();
})
.then(function (value2) {
    ...
})
```

6. Error handling

As mentioned previously, whatever you return in an error handler becomes a fulfillment value (not rejection value!). That allows you to specify default values that are used in case of failure:

```
retrieveFileName()
.catch(function () {
    // Something went wrong, use a default value
    return 'Untitled.txt';
})
.then(function (fileName) {
    ...
});
```

6.1. Catching exceptions

Exceptions in the executor are passed on to the next error handler.

```
new Promise(function (resolve, reject) {
    throw new Error();
})
.catch(function (err) {
    // Handle error here
});
```

As are exceptions that are thrown in either one of `then`’s parameters:

```
asyncFunc()
.then(function (value) {
    throw new Error();
})
.catch(function (reason) {
    // Handle error here
});
```

amazon (3)
asmjs (3)
back to the future (3)
bitwise_ops (3)
css (3)
es2016 (3)
flattr (3)
fluentconf (3)
food (3)
foreign languages (3)
house (3)
icloud (3)
info mgmt (3)
jsfuture (3)
jsstyle (3)
linux (3)
mozilla (3)
python (3)
regexp (3)
samsung (3)
tizen (3)
traffic (3)
typedjs (3)
unix (3)
adobe (2)
angry birds (2)
angularjs (2)
astronomy (2)
audio (2)
comic (2)
design (2)
dom (2)
ecommerce (2)
eval (2)
exploring es6 (2)
facebook flow (2)
facets (2)
flash (2)
free (2)
futurama (2)
guide (2)
history (2)
hyena (2)
internet explorer (2)
iteration (2)
journalism (2)
jquery (2)
jsengine (2)
jslib (2)
law (2)
lightning (2)
markdown (2)
math (2)
meego (2)

6.2. Chaining errors

There can be one or more `then()` method calls that don’t provide an error handler. Then the error is passed on until there is an error handler.

```
asyncFunc1()
.then(asyncFunc2)
.then(asyncFunc3)
.catch(function (reason) {
    // Something went wrong above
});
```

7. Composition

This section describes how you can compose existing promises to create new ones. We have already encountered one way of composing promises: sequential chaining via `then()`. `Promise.all()` and `Promise.race()` provide additional ways of composing.

7.1. `map()` via `Promise.all()`

One nice thing about promises is that many synchronous tools still work, because promise-based functions return results. For example, you can use the array method `map()`:

```
var fileUrls = [
    'http://example.com/file1.txt',
    'http://example.com/file2.txt'
];
var promisedTexts = fileUrls.map(httpGet);
```

`promisedTexts` is an array of promises. `Promise.all()` takes an array of promises (thenables and other values are converted to promises via `Promise.resolve()`) and, once all of them are fulfilled, it fulfills with an array of their values:

```
Promise.all(promisedTexts)
.then(function (texts) {
    texts.forEach(function (text) {
        console.log(text);
    });
})
.catch(function (reason) {
    // Receives first rejection among the promises
});
```

7.2. Timing out via `Promise.race()`

`Promise.race()` takes an array of promises (thenables and other values are converted to promises via `Promise.resolve()`) and returns a promise P. The first of the input promises that is settled passes its settlement on to the output promise.

As an example, let’s use `Promise.race()` to implement a timeout:

```
Promise.race([
    httpGet('http://example.com/file.txt'),
    delay(5000).then(function () {
        throw new Error('Timed out')
    });
])
.then(function (text) { ... })
.catch(function (reason) { ... });
```

8. Promises are always async

A promise library has complete control over whether results are delivered to promise reactions synchronously (right away) or asynchronously (after the current continuation, the current piece of code, is finished). However, the Promises/A+ specification demands that the latter mode of execution be always used. It states so via the following requirement (2.2.4) for the `then()` method:

month (2)
nike (2)
nokia (2)
npm (2)
programming (2)
raffle (2)
repl (2)
servo (2)
sponsor (2)
steve jobs (2)
travel (2)
typescript (2)
usb (2)
winphone (2)
wwdc (2)
airbender (1)
amdefine (1)
aol (1)
app urls (1)
architecture (1)
atscript (1)
basic income (1)
biology (1)
blink (1)
bluetooth (1)
canada (1)
clip (1)
coding (1)
community (1)
cross-platform (1)
deutsch (1)
diaspora (1)
distributed-social-network (1)
dsl (1)
dvd (1)
dzone (1)
emacs (1)
emberjs (1)
energy (1)
esnext news (1)
esprop (1)
example (1)
facetator (1)
feedback (1)
firefly (1)
firefoxos (1)
fritzbox (1)
german (1)
git (1)
guest (1)
guice (1)
h.264 (1)
home entertainment (1)
hosting (1)

> `onFulfilled` or `onRejected` must not be called until the execution context stack contains only platform code.

That means that you code can rely on run-to-completion semantics (as explained in part 1) and that chaining promises won't starve other tasks of processing time.

9. Cheat sheet: the ECMAScript 6 promise API

This section gives an overview of the ECMAScript 6 promise API, as described in the specification.

9.1. Glossary

The promise API is about delivering results asynchronously. A *promise object* (short: promise) is a stand-in for the result, which is delivered via that object.

States:

- A promise is always in either one of three mutually exclusive states:
  - Before the result is ready, the promise is *pending*.
  - If a result is available, the promise is *fulfilled*.
  - If an error happened, the promise is *rejected*.
- A promise is *settled* if "things are done" (if it is either fulfilled or rejected).
- A promise is settled exactly once and then remains unchanged.

Reacting to state changes:

- *Promise reactions* are callbacks that you register with the promise method `then()`, to be notified of a fulfillment or a rejection.
- A *thenable* is an object that has a promise-style `then()` method. Whenever the API is only interested in being notified of settlements, it only demands thenables.

Changing states: There are two operations for changing the state of a promise. After you have invoked either one of them once, further invocations have no effect.

- *Rejecting* a promise means that the promise becomes rejected.
- *Resolving* a promise has different effects, depending on what value you are resolving with:
  - Resolving with a normal (non-thenable) value fulfills the promise.
  - Resolving a promise P with a thenable T means that P can't be resolved anymore and will now follow T's state, including its fulfillment or rejection value. The appropriate P reactions will get called once T settles (or are called if T is already settled).

9.2. Constructor

The constructor for promises has the following signature:

```
var p = new Promise(executor(resolve, reject))
```

It creates a promise whose behavior is determined by the callback `executor`. It can use its parameters to resolve or reject `p`:

- `resolve(x)` resolves `p` with `x`:
  - If `x` is thenable, its settlement is forwarded to `p` (which includes triggering reactions registered via `then()`).
  - Otherwise, `p` is fulfilled with `x`.
- `reject(e)` rejects `p` with the value `e` (often an instance of `Error`).

9.3. Static methods

All static methods of `Promise` support subclassing: they create new instances via their receiver (think: `new this(...)`) and also access other static methods via it (`this.resolve(...)` versus `Promise.resolve(...)`).

Creating promises

The following two methods create new instances of their receiver (their `this`).

- `Promise.resolve(x)`:

    - If `x` is thenable, it is converted to a promise (an instance of the receiver).
    - If `x` is a promise, it is returned unchanged.
    - Otherwise, return a new instance of the receiver that is fulfilled with `x`.
- `Promise.reject(reason)`: creates a new promise that is rejected with the value `reason`.

Composing promises

Intuitively, the static methods `Promise.all()` and `Promise.race()` compose iterables of promises to a single promise. That is:

- They take an iterable. The elements of the iterable are converted to promises via `this.resolve()`.
- They return a new promise. That promise is a fresh instance of the receiver.

The methods are:

- `Promise.all(iterable)`: returns a promise that…
    - is fulfilled if all elements in `iterable` are fulfilled.
      Fulfillment value: array with fulfillment values.
    - is rejected if any of the elements are rejected.
      Rejection value: first rejection value.
- `Promise.race(iterable)`: the first element of `iterable` that is settled is used to settle the returned promise.

9.4. Instance prototype methods

`Promise.prototype.then(onFulfilled, onRejected)`:

- The callbacks `onFulfilled` and `onRejected` are called *reactions*.
- `onFulfilled` is called immediately if the promise is already fulfilled or as soon as it becomes fulfilled. Similarly, `onRejected` is informed of rejections.
- `then()` returns a new promise Q (created via the constructor of the receiver):
    - If either of the reactions returns a value, Q is resolved with it.
    - If either of the reactions throws an exception, Q is rejected with it.
- Omitted reactions:
    - If `onFulfilled` has been omitted, a fulfillment of the receiver is forwarded to the result of `then()`.
    - If `onRejected` has been omitted, a rejection of the receiver is forwarded to the result of `then()`.

Default values for omitted reactions could be implemented like this:

```
function defaultOnFulfilled(x) {
    return x;
}
function defaultOnRejected(e) {
    throw e;
}
```

`Promise.prototype.catch(onRejected)`:

- Same as `then(null, onRejected)`.

10. Pros and cons of promises

10.1. The pros

Unifying asynchronous APIs

One important advantage of promises is that they will increasingly be used by asnychronous browser APIs and unify currently diverse and incompatible patterns and conventions. Let’s look at two upcoming promise-based APIs.

The fetch API is a promise-based alternative to XMLHttpRequest:

```
fetch(url)
.then(request => request.text())
.then(str => ...)
```

`fetch()` returns a promise for the actual request, `text()` returns a promise for the content as a string.

The ECMAScript 6 API for programmatically importing modules is based on promises, too:

```
System.import('some_module.js')
then(some_module => {
   ...
)
```

Promises versus events

Compared to events, promises are better for handling one-off results. It doesn't matter whether you register for a result before or after it has been computed, you will get it. This advantage of promises is fundamental in nature. On the flip side, you can't use them for handling recurring events. Chaining is another advantage of promises, but one that could be added to event handling.

Promises versus callbacks

Compared to callbacks, promises have cleaner function (or method) signatures. With callbacks, parameters are used for input and output:

```
fs.readFile(name, opts?, function (err, data))
```

With promises, all parameters are used for input:

```
readFilePromisified(name, opts?)
    .then(dataHandler, errorHandler)
```

Additional promise advantages include better error handling (which integrates exceptions) and easier composition (because you can reuse some synchronous tools such as `Array.prototype.map()`).

10.2. The cons

Promises work well for for single asynchronous results. They are not suited for:

- Recurring events: If you are interested in those, take a look at reactive programming, which add a clever way of chaining to normal event handling.
- Streams of data: A standard for supporting those is currently in development.

ECMAScript 6 promises lack two features that are sometimes useful:

- You can't cancel them.
- You can't query them for how far along they are (e.g. to display a progress bar in a client-side user interface).

The Q promise library has support for the latter and there are plans to add both capabilities to Promises/A+.

11. Promises and generators

With the help of a utility function such as `Q.spawn()`, you can use promise-based functions inside shallow coroutines, implemented via generators. This has the important advantage that the code looks synchronous and that you can use synchronous mechanisms such a `try-catch`:

```
Q.spawn(function* () {
    try {
        let [foo, bar] = yield Promise.all([ // (A)
            httpGet('foo.json'),
            httpGet('bar.json')
        ]);
        render(foo);
```

```
            render(bar);
        } catch (e) {
            console.log('Read failed: ' + e);
        }
    });
```

The parameter of `Q.spawn()` is a generator function [7]. If the `yield` operator is used, the following things happen:

1. Execution of the function is paused.
2. The operand of `yield` is "returned" by the function. (It's not exactly a "return", but ignore that for now.)
3. Later, the function can be resumed with a value or an exception. In the former case, execution continues where it was previously paused and `yield` returns the value. In the latter case, an exception is thrown inside the function, as if it were thrown "inside" `yield`'s operand.

Thus, it's clear what `Q.spawn()` has to do: When the generator function yields a promise, `spawn` registers reactions and waits for a settlement. If the promise is fulfilled, the generator is resumed with the result. If the promise is rejected, an exception is thrown inside the generator.

There is a proposal to add support for spawning to JavaScript, via the new syntactic construct "async functions". The previous example as an async function looks as follows. Under the hood, there is not much of a difference – async functions are based on generators.

```
async function () {
    try {
        let [foo, bar] = await Promise.all([
            httpGet('foo.json'),
            httpGet('bar.json')
        ]);
        render(foo);
        render(bar);
    } catch (e) {
        console.log('Read failed: ' + e);
    }
}
```

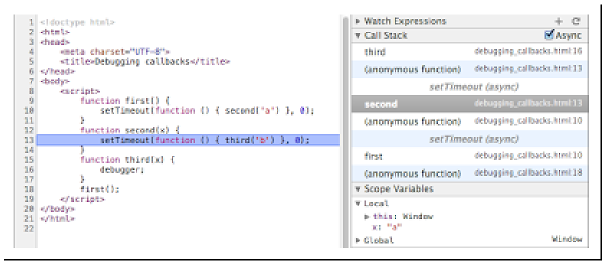
12. Debugging promises

The main challenge with debugging asynchronous code is that it contains asynchronous function and method calls. Asynchronous calls originate in one task and are carried out in a new task. If something goes wrong in the new task, a stack trace will only cover that task and not contain information about previous tasks. Thus, you have to make do with much less debugging information in asynchronous programming.

Google Chrome recently got the ability to debug asynchronous code [6]. It doesn't completely support promises, yet, but it's impressive how well it handles normal asynchronous calls. For example, in the following code, `first` asynchronously calls `second` which in turn calls `third`.

```
function first() {
    setTimeout(function () { second('a') }, 0); // (A)
}
function second(x) {
    setTimeout(function () { third('b') }, 0); // (B)
}
function third(x) {
    debugger;
}
first();
```

As you can see in the screen shot, the debugger shows a stack trace that contains all three functions. It even includes the anonymous functions in line (A) and (B).



```
<!doctype html>
<html>
<head>
    <meta charset="UTF-8">
    <title>Debugging callbacks</title>
</head>
<body>
    <script>
        function first() {
            setTimeout(function () { second('a') }, 0);
        }
        function second(x) {
            setTimeout(function () { third('b') }, 0);
        }
        function third(x) {
            debugger;
        }
        first();
    </script>
</body>
</html>
```

```
Call Stack
third                   debugging_callbacks.html:16
(anonymous function)    debugging_callbacks.html:13
          setTimeout (async)
second                  debugging_callbacks.html:13
(anonymous function)    debugging_callbacks.html:10
          setTimeout (async)
first                   debugging_callbacks.html:10
(anonymous function)    debugging_callbacks.html:18
Scope Variables
Local
  this: Window
  x: "a"
Global                  Window
```

13. The internals of promises

In this section, we will approach promises from a different angle: Instead of learning how to use the API, we will look at a simple implementation of it. This different angle helped me greatly with making sense of promises.

The promise implementation is called DemoPromise and available on GitHub. In order to be easier to understand, it doesn't completely match the API. But it is close enough to still give you much insight into the challenges that actual implementations are facing.

`DemoPromise` is a constructor with three instance prototype methods:

- `DemoPromise.prototype.resolve(value)`
- `DemoPromise.prototype.reject(reason)`
- `DemoPromise.prototype.then(onFulfilled, onRejected)`

That is, `resolve` and `reject` are methods (versus functions handed to a callback parameter of the constructor).

13.1. A stand-alone promise

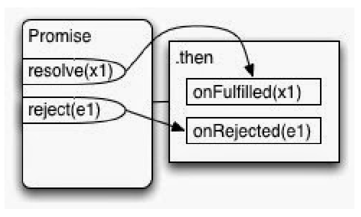
Our first implementation is a stand-alone promise with minimal functionality:

- You can create a promise.
- You can resolve or reject a promise and you can only do it once.
- You can register *reactions* (callbacks) via `then()`. The method does not support chaining, yet – it does not return anything. It must work independently of whether the promise has already been settled or not.

This is how this first implementation is used:

```
var dp = new DemoPromise();
dp.resolve('abc');
dp.then(function (value) {
    console.log(value); // abc
});
```

The following diagram illustrates how our first `DemoPromise` works:



Let's examine `then()` first. It has to handle two cases:

- If the promise is still pending, it queues invocations of `onFulfilled` and `onRejected`, to be used when the promise is settled.
- If the promise is already fulfilled or rejected, `onFulfilled` or `onRejected` can be invoked right away.

```
DemoPromise.prototype.then = function (onFulfilled, onRejected) {
    var self = this;
    var fulfilledTask = function () {
        onFulfilled(self.promiseResult);
    };
    var rejectedTask = function () {
        onRejected(self.promiseResult);
    };
    switch (this.promiseState) {
        case 'pending':
            this.fulfillReactions.push(fulfilledTask);
            this.rejectReactions.push(rejectedTask);
            break;
        case 'fulfilled':
            addToTaskQueue(fulfilledTask);
            break;
        case 'rejected':
            addToTaskQueue(rejectedTask);
            break;
    }
};
function addToTaskQueue(task) {
    setTimeout(task, 0);
}
```

`resolve()` works as follows: If the promise is already settled, it does nothing (ensuring that a promise can only be settled once). Otherwise, the state of the promise changes to `'fulfilled'` and the result is cached in `this.promiseResult`. All fulfillment reactions that have been enqueued so far must be triggered now.

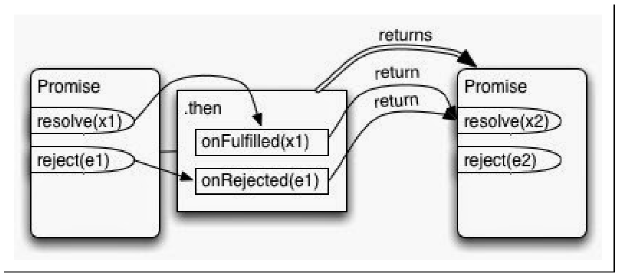
```
    Promise.prototype.resolve = function (value) {
        if (this.promiseState !== 'pending') return;
        this.promiseState = 'fulfilled';
        this.promiseResult = value;
        this._clearAndEnqueueReactions(this.fulfillReactions);
        return this; // enable chaining
    };
    Promise.prototype._clearAndEnqueueReactions = function (reactions) {
        this.fulfillReactions = undefined;
        this.rejectReactions = undefined;
        reactions.map(addToTaskQueue);
    };
```

`reject()` is similar to `resolve()`.

13.2. Chaining

The next feature we implement is chaining:

- `then()` returns a promise that is resolved with what either `onFulfilled` or `onRejected` return.
- If `onFulfilled` or `onRejected` are missing, whatever they would have received is passed on to the promise returned by `then()`.



Obviously, only `then()` changes:

```
    DemoPromise.prototype.then = function (onFulfilled, onRejected) {
        var returnValue = new DemoPromise(); // (A)
        var self = this;

        var fulfilledTask;
        if (typeof onFulfilled === 'function') {
            fulfilledTask = function () {
                var r = onFulfilled(self.promiseResult);
                returnValue.resolve(r); // (B)
            };
        } else {
            fulfilledTask = function () {
                returnValue.resolve(self.promiseResult); // (C)
            };
        }

        var rejectedTask;
        if (typeof onRejected === 'function') {
            rejectedTask = function () {
                var r = onRejected(self.promiseResult);
                returnValue.resolve(r); // (D)
            };
        } else {
            rejectedTask = function () {
                // Important: we must reject here!
                // Normally, result of `onRejected` is used to resolve
                returnValue.reject(self.promiseResult); // (E)
            };
        }
        ...
        return returnValue; // (F)
    };
```

`then()` creates and returns a new promise (lines (A) and (F)). Additionally, `fulfilledTask` and `rejectedTask` are set up differently: After a settlement...

- The result of `onFulfilled` is used to resolve `returnValue` (line (B)).
    - If `onFulfilled` is missing, we use the fulfillment value to resolve `returnValue` (line (C)).
- The result of `onRejected` is used to resolve (not reject!) `returnValue` (line (D)).
    - If `onRejected` is missing, we use the rejection value to reject `returnValue` (line (E)).

13.3. Flattening

Flattening is mostly about making chaining more convenient: Normally, returning a value from a reaction passes it on to the next `then()`. If we return a promise, it would be nice if it could be “unwrapped” for us, like in the following example:

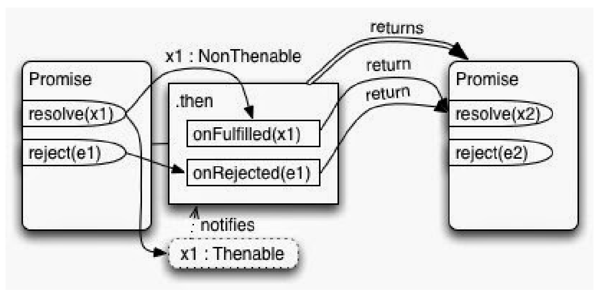
```
    asyncFunc1()
    .then(function (value1) {
        return asyncFunc2(); // (A)
    })
    .then(function (value2) {
        // value2 is fulfillment value of asyncFunc2() promise
        console.log(value2);
    });
```

We returned a promise in line (A) and didn’t have to nest a call to `then()` inside the current method, we could invoke `then()` on the method’s result. Thus: no nested `then()`, everything remains flat.

We implement this by letting the `resolve()` method do the flattening:

- Resolving a promise P with a promise Q means that Q’s settlement is forwarded to P’s reactions.
- P becomes “locked in” on Q: it can’t be resolved (incl. rejected), anymore. And its state and result are always the same as Q’s.

We can make flattening more generic if we allow Q to be a thenable (instead of only a promise).



To implement locking-in, we introduce a new boolean flag `this.alreadyResolved`. Once it is true, `this` is locked and can't be resolved anymore. Note that `this` may still be pending, because its state is now the same as the promise it is locked in on.

```
DemoPromise.prototype.resolve = function (value) {
    if (this.alreadyResolved) return;
    this.alreadyResolved = true;
    this._doResolve(value);
    return this; // enable chaining
};
```

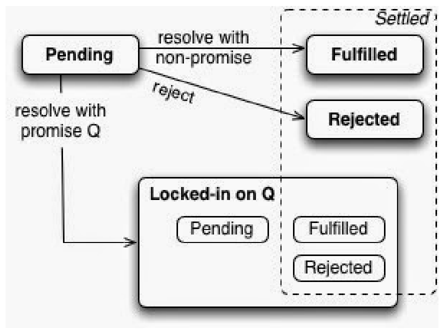
The actual resolution now happens in the private method `_doResolve()`:

```
DemoPromise.prototype._doResolve = function (value) {
    var self = this;
    // Is `value` a thenable?
    if (value !== null && typeof value === 'object'
        && 'then' in value) {
        addToTaskQueue(function () { // (A)
            value.then(
                function onFulfilled(value) {
                    self._doResolve(value);
                },
                function onRejected(reason) {
                    self._doReject(reason);
                });
        });
    } else {
        this.promiseState = 'fulfilled';
        this.promiseResult = value;
        this._clearAndEnqueueReactions(this.fulfillReactions);
    }
};
```

The flattening is performed in line (A): If `value` is fulfilled, we want `self` to be fulfilled and if `value` is rejected, we want `self` to be rejected. The forwarding happens via the private methods `_doResolve` and `_doReject`, to get around the protection via `alreadyResolved`.

13.4. Promise states in more detail

With chaining, the states of promises become more complex (as covered by Sect. 25.4 of the ECMAScript 6 specification):



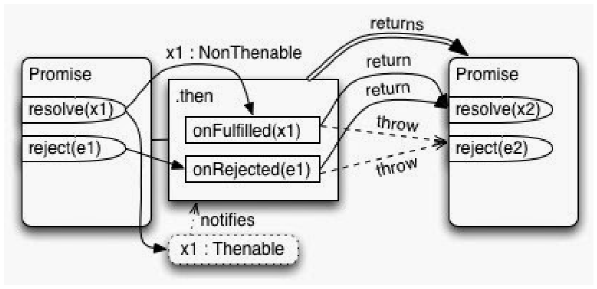
If you are only *using* promises, you can normally adopt a simplified worldview and ignore locking-in. The most important state-related concept remains “settledness”: a promise is settled if it is either fulfilled or rejected. After a promise is settled, it doesn’t change, anymore (state and fulfillment or rejection value).

If you want to *implement* promises then “resolving” matters, too and is now harder to understand:

- Intuitively, “resolved” means “can’t be (directly) resolved anymore”. A promise is resolved if it is either settled or locked in. Quoting the spec: “An unresolved promise is always in the pending state. A resolved promise may be pending, fulfilled or rejected.”
- Resolving does not necessarily lead to settling: you can resolve a promise with another one that is always pending.
- Resolving now includes rejecting (i.e., it is more general): you can reject a promise by resolving it with a rejected promise.

13.5. Exceptions

As our final feature, we’d like our promises to handle exceptions in user code as rejections. For now, “user code” means the two callback parameters of `then()`.

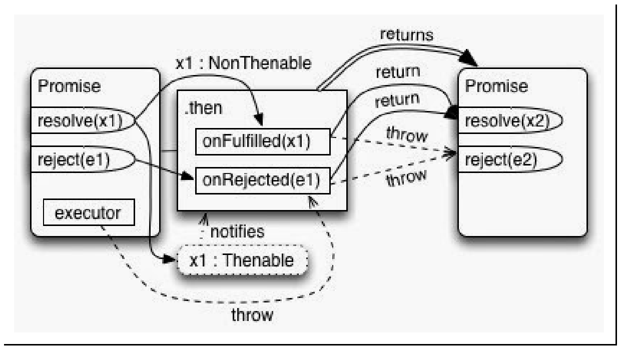


The following excerpt shows how we turn exceptions inside `onFulfilled` into rejections – by wrapping a `try-catch` around its invocation in line (A).

```
var fulfilledTask;
if (typeof onFulfilled === 'function') {
    fulfilledTask = function () {
        try {
            var r = onFulfilled(self.promiseResult); // (A)
            returnValue.resolve(r);
        } catch (e) {
            returnValue.reject(e);
        }
    };
} else {
    fulfilledTask = function () {
        returnValue.resolve(self.promiseResult);
    };
}
```

13.6. Revealing constructor pattern

If we wanted to turn `DemoPromise` into an actual promise implementation, we’d still need to implement the revealing constructor pattern [5]: ES6 promises are not resolved and rejected via methods, but via functions that are handed to the *executor*, the callback parameter of the constructor.



If the executor throws an exception then “its” promise must be rejected.

14. Two useful additional promise methods

This section describes two useful methods that are easy to add to ES6 promises. Many of the more comprehensive promise libraries have them.

14.1. `done()`

When you chain several promise method calls, you risk silently discarding errors. For example:

```
function doSomething() {
    asyncFunc()
    .then(f1)
    .catch(r1)
    .then(f2); // (A)
}
```

If `then()` in line (A) produces a rejection, it will never be handled anywhere. The promise library Q provides a method `done()`, to be used as the last element in a chain of method calls. It either replaces the last `then()` (and has one to two arguments):

```
function doSomething() {
    asyncFunc()
    .then(f1)
    .catch(r1)
    .done(f2);
}
```

Or it is inserted after the last `then()` (and has zero arguments):

```
function doSomething() {
    asyncFunc()
    .then(f1)
    .catch(r1)
    .then(f2)
    .done();
}
```

Quoting the Q documentation:

> The Golden Rule of `done` vs. `then` usage is: either return your promise to someone else, or if the chain ends with you, call `done` to terminate it. Terminating with `catch` is not sufficient because the catch handler may itself throw an error.

This is how you would implement `done()` in ECMAScript 6:

```
Promise.prototype.done = function (onFulfilled, onRejected) {
    this.then(onFulfilled, onRejected)
    .catch(function (reason) {
        setTimeout(() => { throw reason }, 0);
    });
};
```

While done's functionality is clearly useful, it has not been added to ECMAScript 6, because this kind of check can be performed automatically by debuggers in the future (a thread on es-discuss provides further background).

14.2. `finally()`

Sometimes you want to perform an action independently of whether an error happened or not. For example, to clean up after you are done with a resource. That's what the promise method `finally()` is for, which works much like the `finally` clause in exception handling. Its callback receives no arguments, but is notified of either a resolution or a rejection.

```
createResource(...)
.then(function (value1) {
    // Use resource
})
.then(function (value2) {
    // Use resource
})
.finally(function () {
    // Clean up
});
```

This is how `Domenic Denicola` proposes to implement `finally()`:

```
Promise.prototype.finally = function (callback) {
    let p = this.constructor;
    // We don't invoke the callback in here,
    // because we want then() to handle its exceptions
    return this.then(
        // Callback fulfills: pass on predecessor settlement
        // Callback rejects: pass on rejection (=omit 2nd arg.)
        value  => p.resolve(callback()).then(() => value),
        reason => p.resolve(callback()).then(() => { throw reason })
    );
};
```

The callback determines how the settlement of the receiver (`this`) is handled:

- If the callback throws an exception or returns a rejected promise then that becomes/contributes the rejection value.
- Otherwise, the settlement of the receiver becomes the settlement of the promise returned by `finally()`. In a way, we take `finally()` out of the chain of methods.

Example 1 (by Jake Archibald): using `finally()` to hide a spinner. Simplified version:

```
showSpinner();
fetchGalleryData()
.then(data => updateGallery(data))
.catch(showNoDataError)
.finally(hideSpinner);
```

Example 2 (by Kris Kowal): using `finally()` to tear down a test.

```
var HTTP = require("q-io/http");
var server = HTTP.Server(app);
return server.listen(0)
.then(function () {
    // run test
})
.finally(server.stop);
```

15. ES6-compatible promise libraries

There are many promise libraries out there. The following ones conform to the ECMAScript 6 API, which means that you can use them now and easily migrate to native ES6 later.

- "RSVP.js" by Stefan Penner is a superset of the ES6 promise API.

    - "ES6-Promises" by Jake Archibald extracts just the ES6 API out of RSVP.js.
- "Native Promise Only (NPO)" by Kyle Simpson is "a polyfill for native ES6 promises, as close as possible (no extensions) to the strict spec definitions".
- "Lie" by Calvin Metcalf is "a small, performant, promise library implementing the Promises/A+ spec".
- `Q.Promise` by Kris Kowal implements the ES6 API.
- Lastly, the "ES6 Shim" by Paul Millr includes `Promise`.

16. Interfacing with legacy asynchronous code

When you are using a promise library, you sometimes need to use non-promise-based asynchronous code. This section explains how to do that for Node.js-style asynchronous functions and jQuery deferreds.

16.1. Interfacing with Node.js

The promise library Q has several tool functions for converting functions that use Node.js-style `(err,result)` callbacks to ones that return a promise (there are even functions that do the opposite – convert promise-based functions to ones that accept callbacks). For example:

```
var readFile = Q.denodeify(FS.readFile);

readFile('foo.txt', 'utf-8')
.then(function (text) {
    ...
});
```

denodify is a micro-library that only provides the "nodification" functionality and complies with the ECMAScript 6 promise API.

16.2. Interfacing with jQuery

jQuery has deferreds which are similar to promises, but have several differences that prevent compatibility. Their method `then()` is almost like that of ES6 promises (main difference: it doesn't catch errors in reactions). Thus, we can convert a jQuery deferred to an ES6 promise via `Promise.resolve()`:

```
Promise.resolve(
    jQuery.ajax({
        url: 'somefile.html',
        type: 'GET'
    }))
.then(function (data) {
    console.log(data);
})
.catch(function (reason) {
    console.error(reason);
});
```

17. Further reading

- [1]: "Promises/A+", edited by Brian Cavalier and Domenic Denicola (the de-facto standard for JavaScript promises)
- [2]: "JavaScript Promises: There and back again" by Jake Archibald (good general intro to promises)
- [3]: "Promise Anti-Patterns" by Tao of Code (tips and techniques)
- [4]: "Promise Patterns" by Forbes Lindesay
- [5]: "The Revealing Constructor Pattern" by Domenic Denicola (this pattern is used by the `Promise` constructor)
- [6]: "Debugging Asynchronous JavaScript with Chrome DevTools" by Pearl Chen
- [7]: Iterators and generators in ECMAScript 6

12 Comments The 2ality blog 1 Login

Recommend 6 Share Sort by Best

Join the discussion…

Benjamin Gruenbaum • 2 years ago

In addition, it's worth mentioning Bluebird promises - which are faster than native promises at the moment by a considerable amount - include long async stack traces, a much richer API and automatic "promisification" for NodeJS.

In NodeJS projects I always opt for them.

4 • Reply • Share ›

Roy Ling • 2 years ago

I think the constructor signature - var p = new Promise(executor(resolve, reject)) is confusing, after all the executor parameter is a function expression instead of function call. It would be better to say: var p = new Promise(function executor(resolve, reject) { ... }). Right?

1 • Reply • Share ›

Axel Rauschmayer Mod ➦ Roy Ling • 2 years ago

I'll probably switch to TypeScript's notation eventually.

• Reply • Share ›

Benjamin Gruenbaum • 2 years ago

Good post, here's some shameless (self) promotion with some further reading on stuff you didn't really address:

The deferred anti pattern - really common: http://stackoverflow.com/quest...

How to convert a callback API to promises: http://stackoverflow.com/quest...

What are promises and how should I use them (written by Petka): https://github.com/petkaantono...

Problems inherent to jQuery promises: http://stackoverflow.com/quest...

1 • Reply • Share ›

Artin • 2 years ago

I've just started using Promises and faced one issue that was not obvious for me when I was reading about it.

Promises eat my exceptions!

I'm talking about unhandled Promise rejections. With callbacks I'm pretty sure whatever goes wrong - I'll see that in console. With Promises I can forget (or just do not want) to add .catch() block, and it'll just silently die inside Promise.

Is there any cross-environment (browser, server) way to catch that rejections?

• Reply • Share ›

Artin ➦ Artin • 2 years ago

Looks like uncaught rejections is not a problem in lovely Chrome.

Unlike polyfill promises in nodejs (bluebird), native Chrome promises just drop errors to the console:

```
Uncaught (in promise) Error: My error
    at Object.func (source.js:28:15)
at Object.func (source.js:31:29)
at source.js:52:31
at GeneratorFunctionPrototype.next (native)
at onFulfilled (co.js:68:23)
```

• Reply • Share ›

Bob Myers • 2 years ago

Why does this fail in Babel?

class PersistedPromise extends Promise { }
PersistedPromise.resolve(1);

With the command `babel-node --experimental promise.js`, it yields

Promise.apply(this, arguments);
^
TypeError: [object Object] is not a promise

which is a line in the generated constructor.

Also fails in traceur with a slightly different error message about something not being a Promise.

• Reply • Share ›

Axel Rauschmayer Mod → Bob Myers • 2 years ago

This may be a Chrome problem (not an issue in, e.g., babel-node running on an older Node.js). There, Promise.resolve is a native function. Babel may be able to work around this, consider submitting an issue: https://github.com/babel/babel...

• Reply • Share ›

Yo Dirkx • 2 years ago

I think in example 6.2, you're immediately executing asyncFunc2 and asyncFunc3. Shouldn't you pass the function to then(), so it waits execution until asyncFunc1 is done?

http://jsfiddle.net/rudiedirkx...

• Reply • Share ›

Axel Rauschmayer Mod → Yo Dirkx • 2 years ago

True! Fixed now, thanks.

• Reply • Share ›

CYB • 2 years ago

There's a wrong link of **Debugging Asynchronous JavaScript with Chrome DevTools by Pearl Chen**

• Reply • Share ›

Axel Rauschmayer Mod → CYB • 2 years ago

Fixed, thanks!

• Reply • Share ›

Subscribe **Add Disqus to your site Add Disqus Add** **Privacy**

Newer Post Home Older Post

Subscribe to: Post Comments (Atom)