Patterns

by Forbes Lindesay

Introduction

We've seen how even just doing two simple operations one after another can get extremely complex when considering error handling in asynchronous code. We've also seen how promises help you mitigate this via `.then`, which causes errors to bubble up the stack by default.

In this article we'll cover some of the more advanced patterns for promise use and some of the helper methods to make your Promise code more concise.

Promise.resolve(value)

Sometimes you already have a value and you want to convert it into a promise. You may also sometimes find yourself with a value that may or may not be a promise. Finally you might find you have a value that is a promise but does not work as it should (e.g. a jQuery promise) and want to convert it into a true promise.

```
var value = 10;
var promiseForValue = Promise.resolve(value);
// equivalent to
var promiseForValue = new Promise(function (fulfill) {
  fulfill(value);
});
```

```
var jQueryPromise = $.ajax('/ajax-endpoint');
var realPromise = Promise.resolve(jQueryPromise);
// equivalent to
var realPromise = new Promise(function (fulfill, reject) {
  jQueryPromise.then(fulfill, reject);
});
```

```
var maybePromise = Math.random() > 0.5 ? 10 : Promise.resolve(10);
var definitelyPromise = Promise.resolve(maybePromise);
// equivalent to
var definitelyPromise = new Promise(function (fulfill, reject) {
  if (isPromise(maybePromise)) {
    maybePromise.then(fulfill, reject);
  } else {
    fulfill(maybePromise);
  }
});
```

Promise.reject

It's best to always avoid throwing synchronous exceptions in an asychronous method. Always returning a promise has the benefit that people can always handle all errors in the same consistent way. To make it easier to do this, there is a shortcut for generating a rejected promise.

```
var rejectedPromise = Promise.reject(new Error('Whatever'));
// equivalent to
var rejectedPromise = new Promise(function (fulfill, reject) {
  reject(new Error('Whatever'));
});
```

Parallel operations

Trying to do this in parallel only gets more complicated. Consider the following function which attempts to read an array of files (specified by filename) and parse them as JSON then returns the resulting array via a callback:

```
function readJsonFiles(filenames, callback) {
  var pending = filenames.length;
  var called = false;
  var results = [];
  if (pending === 0) {
    // we need to return early in the case where there
    // are no files to read, but we must not return immediately
    // because that unleashes "Zalgo". This makes code very hard
    // to reason about as the order becomes increasingly
    // non-deterministic.
    return setTimeout(function () { callback(); }, 0);
  }
  filenames.forEach(function (filename, index) {
    readJSON(filename, function (err, res) {
      if (err) {
        if (!called) callback(err);
        return;
      }
      results[index] = res;
      if (0 === --pending) {
        callback(null, res);
      }
    });
  });
}
```

That's a maddening amount of code to have to write for such a simple asynchronous function. It is possible to write most of this into a library function that lets you do an asynchronous "map" operation, but that only solves the very specific case, and can still be remarkably fiddly.

Promise.all

The `all` function returns a new promise which is fulfilled with an array of fulfillment values for the passed promises or rejects with the reason of the first promise that rejects.

```
function readJsonFiles(filenames) {
  // N.B. passing readJSON as a function, not calling it with `()`
  return Promise.all(filenames.map(readJSON));
}
readJsonFiles(['a.json', 'b.json']).done(function (results) {
  // results is an array of the values stored in a.json and b.json
}, function (err) {
  // If any of the files fails to be read, err is the first error
});
```

`Promise.all` is a builtin method, so you don't need to worry about implementing it yourself, but it serves as a nice demo of how easy promises are to work with.

```
function all(promises) {
  var accumulator = [];
  var ready = Promise.resolve(null);

  promises.forEach(function (promise, ndx) {
    ready = ready.then(function () {
      return promise;
    }).then(function (value) {
      accumulator[ndx] = value;
    });
  });

  return ready.then(function () { return accumulator; });
}
```

What's going on here is that we start by creating a variable to store the result (called `accumulator`) and a variable to denote whether the result is up to date (called `ready`). We wait on `ready`, and also update it with each turn of the loop. This leads to us putting each `value` onto the `accumulator` array one at a time in order. By the end of the loop, `ready` is a promise that will wait for all the items to be inserted into the `accumulator` array.

All we have to do at the end is wait for the `ready` promise and then return `accumulator`.

The native implementation will be more efficient than this, but it should give you an idea of how promises can be combined in interesting ways.

Promise.race

Sometimes it is useful to race two promises against each other. Consider the case of writing a timeout function. You could do something like this:

```
function delay(time) {
  return new Promise(function (fulfill) {
    setTimeout(fulfill, time);
  });
}
function timeout(promise, time) {
  return new Promise(function (fulfill, reject) {
    // race promise against delay
    promise.then(fulfill, reject);
    delay(time).done(function () {
      reject(new Error('Operation timed out'));
    });
  });
}
```

Promise.race makes races like this even easier to run:

```
function timeout(promise, time) {
  return Promise.race([promise, delay(time).then(function () {
    throw new Error('Operation timed out');
  })]);
}
```

Whichever promise settles (fulfills or rejects) first wins the race, and determines the result.

Further Reading

- Generators (/generators/) - learn how to use generators and promises together to make programming with promises really easy (in browsers with ES6 support)
- MDN (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise) - The mozilla developer network has great documentation on promises.
- then/promise (https://github.com/then/promise) - An implementation of all these helper methods in JavaScript.

← API Reference (/api/)

generators → (/generators/)

Developed by @ForbesLindesay (http://www.forbeslindesay.co.uk)
Can you make this better? Please fork it on GitHub (https://github.com/ForbesLindesay/promisejs.org)