Wrangle Async Tasks with jQuery Promises

Trevor Burnham on Feb 22nd 2012 with 12 comments

Tutorial Details

-
- **Topic:** jQuery Deferreds
- **Difficulty:** Moderate

Promises are an exciting jQuery feature that make it a breeze to manage async events. They allow you to write clearer, shorter callbacks and keep high-level application logic separate from low-level behaviors.

Once you understand Promises, you'll want to use them for everything from AJAX calls to UI flow. That's a promise!

Understanding Promises

> Once a Promise is resolved or rejected, it'll remain in that state forever.

A Promise is an object that represents a one-time event, typically the outcome of an async task like an AJAX call. At first, a Promise is in a *pending* state. Eventually, it's either *resolved* (meaning the task is done) or *rejected* (if the task failed). Once a Promise is resolved or rejected, it'll remain in that state forever, and its callbacks will never fire again.

You can attach callbacks to the Promise, which will fire when the Promise is resolved or rejected. And you can add more callbacks whenever you want – even after the Promise has been resolved/rejected! (In that case, they'll fire immediately.)

Plus, you can combine Promises logically into new Promises. That makes it trivially easy to write code that says, "When all of these things have happened, do this other thing."

And that's all you need to know about Promises in the abstract. There are several JavaScript implementations to choose from. The two most notable are Kris Kowal's q, based on the CommonJS Promises/A spec, and jQuery Promises (added in jQuery 1.5). Because of jQuery's ubiquity, we"ll use its implementation in this tutorial.

Making Promises with $.Deferred

Every jQuery Promise begins with a Deferred. A Deferred is just a Promise with methods that allow its owner to resolve or reject it. All other Promises are "read-only" copies of a Deferred; we'll talk about those in the next section. To create a Deferred, use the `$.Deferred()` constructor:

> A Deferred is just a Promise with methods that allow its owner to resolve or reject it.

```
1 var deferred = new $.Deferred();
2
```

```
deferred.state();  // "pending"
deferred.resolve();
deferred.state();  // "resolved"
deferred.reject(); // no effect, because the Promise was already resolved
```

(**Version note:** `state()` was added in jQuery 1.7. In 1.5/1.6, use `isRejected()` and `isResolved()`.)

We can get a "pure" Promise by calling a Deferred's `promise()` method. The result is identical to the Deferred, except that the `resolve()` and `reject()` methods are missing.

```
var deferred = new $.Deferred();
var promise = deferred.promise();

promise.state();  // "pending"
deferred.reject();
promise.state();  // "rejected"
```

The `promise()` method exists purely for encapsulation: If you return a Deferred from a function, it might be resolved or rejected by the caller. But if you only return the pure Promise corresponding to that Deferred, the caller can only read its state and attach callbacks. jQuery itself takes this approach, returning pure Promises from its AJAX methods:

```
var gettingProducts = $.get("/products");

gettingProducts.state();  // "pending"
gettingProducts.resolve;  // undefined
```

Using the `-ing` tense in the name of a Promise makes it clear that it represents a process.

Modeling a UI Flow With Promises

Once you have a Promise, you can attach as many callbacks as you like using the `done()`, `fail()`, and `always()` methods:

```
promise.done(function() {
  console.log("This will run if this Promise is resolved.");
});

promise.fail(function() {
  console.log("This will run if this Promise is rejected.");
});

promise.always(function() {
  console.log("And this will run either way.");
});
```

Version Note: `always()` was referred to as `complete()` before jQuery 1.6.

There's also a shorthand for attaching all of these types of callbacks at once, `then()`:

```
promise.then(doneCallback, failCallback, alwaysCallback);
```

Callbacks are guaranteed to run in the order they were attached in.

One great use case for Promises is representing a series of potential actions by the user. Let's take a basic AJAX form, for example. We want to ensure that the form can only be submitted once, and that the user receives some acknowledgement when they submit the form. Furthermore, we want to keep the code describing the application's behavior separate from the code that touches the page's markup. This will make unit testing much easier, and minimize the amount of code that needs to be changed if we modify our page layout.

```
// Application logic
var submittingFeedback = new $.Deferred();

submittingFeedback.done(function(input) {
  $.post("/feedback", input);
});

// DOM interaction
$("#feedback").submit(function() {
```

```
    submittingFeedback.resolve($("textarea", this).val());

    return false;  // prevent default form behavior
  });
  submittingFeedback.done(function() {
    $("#container").append("<p>Thank you for your feedback!</p>");
  });
```

(We're taking advantage of the fact that arguments passed to `resolve()`/`reject()` are forwarded verbatim to each callback.)

Borrowing Promises From the Future

> `pipe()` returns a new Promise that will mimic any Promise returned from one of the `pipe()` callbacks.

Our feedback form code looks good, but there's room for improvement in the interaction. Rather than optimistically assuming that our POST call will succeed, we should first indicate that the form has been sent (with an AJAX spinner, say), then tell the user whether the submission succeeded or failed when the server responds.

We can do this by attaching callbacks to the Promise returned by `$.post`. But therein lies a challenge: We need to manipulate the DOM from those callbacks, and we've vowed to keep our DOM-touching code out of our application logic code. How can we do that, when the POST Promise is created within an application logic callback?

A solution is to "forward" the resolve/reject events from the POST Promise to a Promise that lives in the outer scope. But how do we do that without several lines of bland boilerplate (`promise1.done(promise2.resolve);`…)? Thankfully, jQuery provides a method for exactly this purpose: `pipe()`.

`pipe()` has the same interface as `then()` (`done()` callback, `reject()` callback, `always()` callback; each callback is optional), but with one crucial difference: While `then()` simply returns the Promise it's attached to (for chaining), `pipe()` returns a new Promise that will mimic any Promise returned from one of the `pipe()` callbacks. In short, `pipe()` is a window into the future, allowing us to attach behaviors to a Promise that doesn't even exist yet.

Here's our *new and improved* form code, with our POST Promise piped to a Promise called `savingFeedback`:

```
// Application logic
var submittingFeedback = new $.Deferred();
var savingFeedback = submittingFeedback.pipe(function(input) {
  return $.post("/feedback", input);
});

// DOM interaction
$("#feedback").submit(function() {
  submittingFeedback.resolve($("textarea", this).val());

  return false;  // prevent default form behavior<br/>
});

submittingFeedback.done(function() {
  $("#container").append("<div class='spinner'>");
});

savingFeedback.then(function() {
  $("#container").append("<p>Thank you for your feedback!</p>");
}, function() {
  $("#container").append("<p>There was an error contacting the server.</p>");
}, function() {
  $("#container").remove(".spinner");
});
```

Finding the Intersection Of Promises

Part of the genius of Promises is their binary nature. Because they have only two eventual states, they can be combined like booleans (albeit booleans whose values may not yet be known).

The Promise equivalent of the logical intersection (AND) is given by `$.when()`. Given a list of Promises, `when()` returns a new Promise that obeys these rules:

1. When **all** of the given Promises are resolved, the new Promise is resolved.
2. When **any** of the given Promises is rejected, the new Promise is rejected.

> Any time you're waiting for multiple unordered events to occur, you should consider using `when()`.

Simultaneous AJAX calls are an obvious use case:

```
$("#container").append("<div class='spinner'>");
$.when($.get("/encryptedData"), $.get("/encryptionKey")).then(function() {
  // both AJAX calls have succeeded
}, function() {
  // one of the AJAX calls has failed
}, function() {
  $("#container").remove(".spinner");
});
```

Another use case is allowing the user to request a resource that may or may not have already be available. For example, suppose we have a chat widget that we're loading with YepNope (see Easy Script Loading with yepnope.js)

```
var loadingChat = new $.Deferred();
yepnope({
  load: "resources/chat.js",
  complete: loadingChat.resolve
});

var launchingChat = new $.Deferred();
$("#launchChat").click(launchingChat.resolve);
launchingChat.done(function() {
  $("#chatContainer").append("<div class='spinner'>");
});

$.when(loadingChat, launchingChat).done(function() {
  $("#chatContainer").remove(".spinner");
  // start chat
});
```

Conclusion

Promises have proven themselves to be an indispensable tool in the ongoing fight against async spaghetti code. By providing a binary representation of individual tasks, they clarify application logic and cut down on state-tracking boilerplate.

> If you'd like to know more about Promises and other tools for preserving your sanity in an ever more asynchronous world, check out my upcoming eBook: Async JavaScript: Recipes for Event-Driven Code (due out in March).

Tags: deferredsjQuerypromises

By Trevor Burnham

I'm the author of *CoffeeScript: Accelerated JavaScript Development*, published by PragProg. I'm a regular conference speaker and available for hire as a CoffeeScript or JavaScript consultant.