Implementing

by Forbes Lindesay

Introduction

This article was originally written as an answer on Stack Overflow (http://stackoverflow.com/questions/23772801/basic-javascript-promise-implementation-attempt/23785244#23785244). The hope is that by seeing how you would go about implementing `Promise` in JavaScript, you may gain a better understanding of how promises behave.

State Machine

Since a promise is just a state machine, we should start by considering the state information we will need later.

```
var PENDING = 0;
var FULFILLED = 1;
var REJECTED = 2;

function Promise() {
  // store state which can be PENDING, FULFILLED or REJECTED
  var state = PENDING;

  // store value or error once FULFILLED or REJECTED
  var value = null;

  // store sucess & failure handlers attached by calling .then or .done
  var handlers = [];
}
```

Transitions

Next, lets consider the two key transitions that can occur, fulfilling and rejecting:

```
var PENDING = 0;
var FULFILLED = 1;
var REJECTED = 2;

function Promise() {
  // store state which can be PENDING, FULFILLED or REJECTED
  var state = PENDING;

  // store value once FULFILLED or REJECTED
  var value = null;

  // store sucess & failure handlers
  var handlers = [];

  function fulfill(result) {
    state = FULFILLED;
    value = result;
  }

  function reject(error) {
    state = REJECTED;
    value = error;
  }
}
```

That gives us the basic low level transitions, but lets consider an extra, higher-level transition called `resolve`

```
var PENDING = 0;
var FULFILLED = 1;
var REJECTED = 2;

function Promise() {
  // store state which can be PENDING, FULFILLED or REJECTED
  var state = PENDING;

  // store value once FULFILLED or REJECTED
  var value = null;

  // store sucess & failure handlers
  var handlers = [];

  function fulfill(result) {
    state = FULFILLED;
    value = result;
  }

  function reject(error) {
    state = REJECTED;
    value = error;
  }

  function resolve(result) {
    try {
      var then = getThen(result);
      if (then) {
        doResolve(then.bind(result), resolve, reject)
        return
      }
      fulfill(result);
    } catch (e) {
      reject(e);
    }
  }
}
```

Note how `resolve` accepts either a promise or a plain value and if it's a promise, waits for it to complete. A promise must never be fulfilled with another promise, so it is this `resolve` function that we will expose, rather than the internal `fulfill`. We've used a couple of helper methods, so lets define those:

```
/**
 * Check if a value is a Promise and, if it is,
 * return the `then` method of that promise.
 *
 * @param {Promise|Any} value
 * @return {Function|Null}
 */
function getThen(value) {
  var t = typeof value;
  if (value && (t === 'object' || t === 'function')) {
    var then = value.then;
    if (typeof then === 'function') {
      return then;
    }
  }
  return null;
}

/**
 * Take a potentially misbehaving resolver function and make sure
 * onFulfilled and onRejected are only called once.
 *
 * Makes no guarantees about asynchrony.
 *
 * @param {Function} fn A resolver function that may not be trusted
 * @param {Function} onFulfilled
 * @param {Function} onRejected
 */
function doResolve(fn, onFulfilled, onRejected) {
  var done = false;
  try {
    fn(function (value) {
      if (done) return
      done = true
      onFulfilled(value)
    }, function (reason) {
      if (done) return
      done = true
      onRejected(reason)
    })
  } catch (ex) {
    if (done) return
    done = true
    onRejected(ex)
  }
}
```

Constructing

We now have the completed internal state machine, but we have yet to expose either a method of resolving the promise or of observing it. Lets start by adding a way of resolving the promise.

```
var PENDING = 0;
var FULFILLED = 1;
var REJECTED = 2;

function Promise(fn) {
  // store state which can be PENDING, FULFILLED or REJECTED
  var state = PENDING;

  // store value once FULFILLED or REJECTED
  var value = null;

  // store sucess & failure handlers
  var handlers = [];

  function fulfill(result) {
    state = FULFILLED;
    value = result;
  }

  function reject(error) {
    state = REJECTED;
    value = error;
  }

  function resolve(result) {
    try {
      var then = getThen(result);
      if (then) {
        doResolve(then.bind(result), resolve, reject)
        return
      }
      fulfill(result);
    } catch (e) {
      reject(e);
    }
  }

  doResolve(fn, resolve, reject);
}
```

As you can see, we re-use `doResolve` because we have another untrusted resolver. The `fn` is allowed to call both `resolve` and `reject` multiple times, and even throw exceptions. It is up to us to ensure that the promise is only resolved or rejected once, and then never transitions into a different state ever again.

Observing (via .done)

We now have a completed state machine, but we still have no way to observe any changes to it. Our ultimate goal is to implement `.then`, but the semantics of `.done` are much simpler so lets implement that first.

Our goal here is to implement `promise.done(onFulfilled, onRejected)` such that:

- only one of `onFulfilled` or `onRejected` is called
- it is only called once
- it is never called until the next tick (i.e. after the `.done` method has returned)
- it is called regardless of whether the promise is resolved before or after we call `.done`

```
var PENDING = 0;
var FULFILLED = 1;
var REJECTED = 2;

function Promise(fn) {
  // store state which can be PENDING, FULFILLED or REJECTED
  var state = PENDING;

  // store value once FULFILLED or REJECTED
  var value = null;

  // store sucess & failure handlers
  var handlers = [];

  function fulfill(result) {
    state = FULFILLED;
    value = result;
    handlers.forEach(handle);
    handlers = null;
  }

  function reject(error) {
    state = REJECTED;
    value = error;
    handlers.forEach(handle);
    handlers = null;
  }

  function resolve(result) {
    try {
      var then = getThen(result);
      if (then) {
        doResolve(then.bind(result), resolve, reject)
        return
      }
      fulfill(result);
    } catch (e) {
      reject(e);
    }
  }

  function handle(handler) {
    if (state === PENDING) {
      handlers.push(handler);
    } else {
      if (state === FULFILLED &&
        typeof handler.onFulfilled === 'function') {
        handler.onFulfilled(value);
      }
      if (state === REJECTED &&
        typeof handler.onRejected === 'function') {
        handler.onRejected(value);
      }
    }
  }

  this.done = function (onFulfilled, onRejected) {
    // ensure we are always asynchronous
    setTimeout(function () {
      handle({
        onFulfilled: onFulfilled,
        onRejected: onRejected
      });
    }, 0);
  }

  doResolve(fn, resolve, reject);
}
```

We make sure to notify the handlers when the Promise is resolved or rejected. We only ever do this in the next tick.

Observing (via .then)

Now that we have `.done` implemented, we can easily implement `.then` to just do the same thing, but construct a new Promise in the process.

```
this.then = function (onFulfilled, onRejected) {
  var self = this;
  return new Promise(function (resolve, reject) {
    return self.done(function (result) {
      if (typeof onFulfilled === 'function') {
        try {
          return resolve(onFulfilled(result));
        } catch (ex) {
          return reject(ex);
        }
      } else {
        return resolve(result);
      }
    }, function (error) {
      if (typeof onRejected === 'function') {
        try {
          return resolve(onRejected(error));
        } catch (ex) {
          return reject(ex);
        }
      } else {
        return reject(error);
      }
    });
  });
}
```

Further Reading

- then/promise (https://github.com/then/promise/blob/master/src/core.js) implements Promise in a very similar way.
- kriskowal/q (https://github.com/kriskowal/q/blob/v1/design/README.md) is a very different implementation of promises and comes with a very nice walkthrough of the design principles behind it.
- petkaantonov/bluebird (https://github.com/petkaantonov/bluebird) is a promise implementation that was designed exclusively for performance (along with its own esoteric helper methods). The Optimization Killers (https://github.com/petkaantonov/bluebird/wiki/Optimization-killers) Wiki page is extremely useful for picking up tips.
- Stack Overflow (http://stackoverflow.com/questions/23772801/basic-javascript-promise-implementation-attempt/23785244#23785244) is the original source of this article.

← 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)