Programming Concurrency in JavaScript with Promises

Darrell Pratt, Architect Leader, GBS

Executive Summary

JavaScript has been regarded as a scripting version of an object oriented programming language, but it has far outpaced other more statically typed languages in the area of concurency and parallelism. The language generally has used the concept of callbacks to allow for parallel execution of tasks, but several libraries and the advent of ECMAScript 6 has brought forward the concept of promises to the language. The core concept to understand in all of JavaScript is that the code be it on the server or in the browser is executed in a single thread. This might seem like an issue, but is actually used to create very high performance code.

Problem Statement

JavaScript is an inheritently a concurrent language as it executes code tied to the clock cycle of the host CPU. The familiar setTimeout() function shows just this. The VM is instructed to sleep for some time and then execute a command at the next cpu cycle after that timeout. This simple function has been extended to a general callback metaphor where most methods in JavaScript behave in a functional manner and take a function as one of their arguments and can then call that function at some point in the future when the return is available. This method has worked for some time and is indeed the common metaphor in NodeJS, but it is rapidly being replaced with the concept of promises. This framework promises to remove the nested entanglement of callbacks and replace it with a more standard concept of concurrent promise objects.

Pyramid of Doom

```
// Code uses JQuery to illustrate the Pyramid of Doom
(function($) {
  $(function(){
    $("button").click(function(e) {
      $.get("/test.json", function(data, textStatus, jqXHR) {
        $(".list").each(function() {
          $(this).click(function(e) {
            setTimeout(function() {
              alert("Hello World!");
            }, 1000);
          });
        });
      });
    });
  });
})(jQuery);
```

The code example above is not too atypical of a JQuery applications logic. If the developer is extremely familiar with this style of coding it doesn't present too many issues, but for anyone else to maintain this code it quickly becomes a nightmare.

Unfortunately, at this point, the callback paradigm has become the standard in many JavaScript libraries. With functions being first class citizens in the language, this programming method is perfectly acceptable, but as seen above, it is quite hard to follow for the developer who first might inherit a code base that is riddled with this style.

Promises as defined in the new ECMAScript 6 specification intend to change this state of coding.

A Promise of Sanity

A promise simply represents an object containing a value or context that might or might not be ready for consumption. On first glance this does not seem to help the developer. One can understand a callback function (meaning, call me, I won't call you).

The promise object basically represents some future state or value return from a long running function. The best example of this is a long running I/O bounded function. Consider a database or remote procedure call that might be blocking to traditional code. By using promises, the calling function can pass a promise that in effect will be fulfilled by the called function when the output is ready to be consumed. This allows code within the calling function to continue executing without blocking the main process.

The basic concept of the promise is that the developer can use an understable set of methods on the object to continue the code execution. Rather than using the nested callback structure as seen above, the developer can instead use methods such as then() and catch(). This allow for chaining of method calls across a series of events and is easy to read for the developer maintaining the code but still gives the application the performance boost of concurrent processing.

Libraries

Whilst ECMAScript 6 does contain a specification for Promises within the standard JS library, several libraries have filled the void while this spec is implemented by browsers and server side JS engines.

Q

Q is described as a tool for making and composing asynchronous promises in JavaScript. This library was written by Kris Kowal, and has become the basis of many tools and frameworks that inherently use asynchronous programming techniques.

It is somewhat easier to understand Q by seeing an example of what it accomplishes. This is shown below.

```
step1(function (value1) {
    step2(value1, function(value2) {
        step3(value2, function(value3) {
            step4(value3, function(value4) {
                // Do something with value4
            });
        });
    });
});
```

```
Q.fcall(promisedStep1)
.then(promisedStep2)
.then(promisedStep3)
.then(promisedStep4)
.then(function (value4) {
    // Do something with value4
})
.catch(function (error) {
    // Handle any error from all above steps
})
.done();
```

This methodology becomes quite important, when the developer wishes to maintain concurrency in the their program but also wants to enforce a sequence of events across some workflow that requires it.

Q provides a rich set of methods in its API to lay a promise framework upon a callback based API. This becomes very useful when writing code in NodeJS which at this point is primarily based upon callbacks. The code examples below show how a developer can wrap callback style functions in promise objects to simplify the interaction with them.

```
function requestOkText(url) {
    var request = new XMLHttpRequest();
    var deferred = Q.defer();

    request.open("GET", url, true);
    request.onload = onload;
    request.onerror = onerror;
    request.onprogress = onprogress;
    request.send();

    function onload() {
        if (request.status === 200) {
            deferred.resolve(request.responseText);
        } else {
            deferred.reject(new Error("Status code was " + request.status));
        }
    }

    function onerror() {
        deferred.reject(new Error("Can't XHR " + JSON.stringify(url)));
    }

    function onprogress(event) {
        deferred.notify(event.loaded / event.total);
    }

    return deferred.promise;
}
```

With this function using a Deferred object, the calling code can then begin to use chaining within the promise object to accomplish it's task in a more straight forward manner. This is seen below.

```
requestOkText("http://localhost:3000")
.then(function (responseText) {
    // If the HTTP response returns 200 OK, log the response text.
    console.log(responseText);
}, function (error) {
    // If there's an error or a non-200 status code, log the error.
    console.error(error);
}, function (progress) {
    // Log the progress as it comes in.
    console.log("Request progress: " + Math.round(progress * 100) + "%");
});
```

At first glance, this is not a simplification, but it doesn indeed lead to more readable code as a result of the sequential view into the asynchronous actions of the program.

When

When is another library of the popular libraries which implements the promises API. The purpose of this library is to once again reduce the nested complexity of asnychronous code with an when this then that methodology. The code example below depicts this.

```
var rest = require('rest');

fetchRemoteGreeting()
    .then(addExclamation)
    .catch(handleError)
    .done(function(greeting) {
        console.log(greeting);
    });

function fetchRemoteGreeting() {
    // returns a when.js promise for 'hello world'
    return rest('http://example.com/greeting');
}

function addExclamation(greeting) {
    return greeting + '!!!!'
}

function handleError(e) {
    return 'drat!';
}
```

Language Path

Promises are coming to the core language in ECMAScript 6. The libraries detailed previously implement the core specification so code written with those should easily port to the core set of libraries when present.

Let's take a step back to why this is important. We have covered the methodology for how one can use a promise library but we haven't completely explained why it is useful to use a Promise. The phrase that a developer new to the world of JavaScript in 2014 might here is non-blocking, asyncronous I/O. This is a very important tenant of NodeJS and more advanced client side JavaScript libraries as well. With JavaScript being a single threaded execution engine, it is important to not block on calls against resources that might take some unknown amount of time. With each cycle of the CPU running and not being used whilst a function blocks for a return, the overall execution of the code is wasting time it might otherwise use more effeciently.

The core JavaScript language will introduce the promise object natively in the next release. This should allow for a universal standard of coding that does not require the callback nesting that is required today. This should make it easier for a non-JavaScript programmer to get a grasp of what is being accomplished within a block of code, but still maintain the general benefits of non-blocking coding standards.

Conclusion

It is best to leave a developer with a bit of code to impress upon them the impact of Promises upon the language. Below is an example of code using callbacks to accomplish a standard task.

```
Parse.User.logIn("user", "pass", {
  success: function(user) {
    query.find({
      success: function(results) {
        results[0].save({ key: value }, {
          success: function(result) {
            // the object was saved.
          }
        });
      }
    });
  }
});
```

This is used throughout the code base of the Buy platform's BI reporting application. The browser will make requests to the server side code and wait for the return JSON object in order to display it to the user in the form of a chart or tabular data.

```
Parse.User.logIn("user", "pass").then(function(user) {
  return query.find();
}).then(function(results) {
  return results[0].save({ key: value });
}).then(function(result) {
  // the object was saved.
});
```

Promises can't quite promise everything to the developer, but the syntax does promise to remove the pyramid of doom from code and allow for asynchronous programming with some sequential readability.