

[Blog Home](#)[Recommended Resources](#)[About](#)[Contact](#)[🔍 Search](#)

FOLLOW:



JAVASCRIPT

NEWER

[Speed Up jQuery Development With CodeLobster](#) >

OLDER

< [Better Backbone Applications With MarionetteJS Giveaway](#)

Apollo GraphQL

**GraphQL at Credit Karma**

Learn how Credit Karma manages GraphQL at scale

RECOMMENDATIONS

[View All Recommendations](#)

Patterns for Asynchronous Programming With Promises

2014/04/23

Promises are currently the best tool we have for asynchronous programming and they appear to be our best hope for the foreseeable future, even if they'll be hiding behind [generators](#) or [async functions](#). For now, we'll need to use promises directly, so we should learn some good techniques for using them right now, especially when dealing with asynchronous operations on



When you purchase anything through any of the above recommended links, you help support this blog. Thank you!



RECENT POSTS

GET STARTED WITH JAVASCRIPT ARRAYS
2016/09/26

collections, whether they happen in parallel or sequentially.

Before We Start

In the code, `asyncOperation` just represents a function that takes a single number parameter, performs an asynchronous operation according to that number, and returns a promise, while `// ...` represents whatever code is specific to your application that operates on the values returned from `asyncOperation`.

Each of the functions I create, it will run the `asyncOperation` on all of the values in the `values` array and return a promise that resolves to an array of the values that `asyncOperation` provides.

Parallel Asynchronous Operations

First we'll take a look at parallel operations. This refers to getting multiple asynchronous operations queued up and running at the same time. By running them in parallel, you can significantly increase your performance. Sadly, this isn't always

COMPOSITION IS KING

2016/08/25

THE COMPLETE-ISH GUIDE TO UPGRADING TO
GULP 4

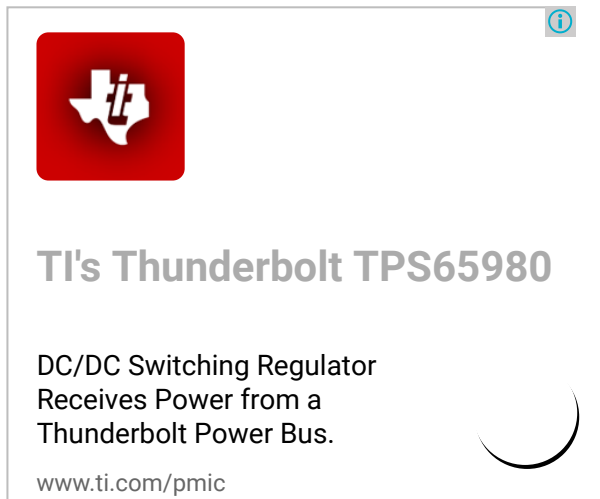
2016/05/25

UNIXSTICKERS REVIEW: WHERE TO FULFILL YOUR
GEEKY NEEDS

2016/04/14

INTEGRATING YOUR DEVELOPMENT WORKFLOW
INTO SUBLIME WITH BUILD SYSTEMS - PART 4:
PROJECT-SPECIFIC BUILDS

2016/03/11



possible. You may be required to run the operations in sequential order, which is what we'll be talking about in the next section.

Anyway, we'll first look at running the asynchronous operations in parallel, but then performing synchronous operations on them in a specific order after all of the asynchronous operations have finished. This gives you a performance boost from the parallel operations, but then brings everything back together to do things in the right order when you need to.

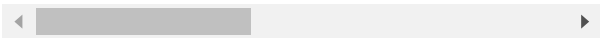
```

1  function parallelAsyncSequence
2      var values = [1,2,3,4];
3
4      // Use `map` to create an array of
5      // `asyncOperation` on each value
6      // They should happen in parallel
7      var operations = values.map(function(v) {
8
9          // Return a promise so we can
10         return Promise.all(operations);
11         // Once all of the operations have finished,
12         // through the results back to the caller
13         newValues.forEach(function(v, i) {
14             // ...
15         });
16
17         // Make sure we return the new values
18         return newValues;
19     });
20 }
```

We use `map` to get all of our asynchronous operations fired off right away, but then use `Promise.all` to wait for them all to finish, and then we just run a loop over the new values and do whatever operations we need to do in the original order.

Sometimes, the order that our synchronous operations run in don't matter. In this case, we can run each of our synchronous operations immediately after their respective asynchronous operations have finished.

```
1  function parallelAsyncUnordered(values, callback) {
2      var values = [1,2,3,4];
3
4      // Use `map` to create an array of asynchronous operations
5      var operations = values.map(function(value) {
6          // return the promise for the asynchronous operation
7          return asyncOperation(value);
8          // ...
9
10         // we want the new value after the asynchronous operation
11         return newValue(value);
12     });
13
14     // return a promise so we can wait for all operations to finish
15     return Promise.all(operations).then(function(values) {
16         // ...
17     });
18 }
```



For this, we use `map` again, but instead of waiting for all of the operations to finish, we provide our own callback to

`map` and do more inside of it. Inside we invoke our asynchronous function and then call `then` on it immediately to set up our synchronous operation to run immediately after the asynchronous one has finished.



Sequential Asynchronous Operations

Let's take a look at some patterns for sequential asynchronous operations. In this case, the first asynchronous operation should finish before moving on to the next asynchronous operation. I have two solutions for doing this, one uses `forEach` and one uses `reduce`. They are quite similar, but the version with `forEach` needs to store a reference to the promise chain, whereas the version with `reduce`

passes it through as the memo.

Essentially, the version with `forEach` is just more explicit and verbose, but they both accomplish the same thing.

```
1  function sequentialAsyncWith
2      var values = [1,2,3,4];
3      var newValues = [];
4      var promise = Promise.resolve()
5
6      values.forEach(function
7          promise = promise.then(
8              return asyncOperation
9          }).then(function(newValues)
10              // ...
11              newValues.push(
12          });
13  });
14
15  return promise.then(function()
16      return newValues;
17  });
18 }
```

```
1  function sequentialAsyncWith
2      var values = [1,2,3,4];
3      var newValues = [];
4
5      return values.reduce(function
6          return memo.then(function
7              return asyncOperation
8          }).then(function(newValues)
9              // ...
10              newValues.push(
11          });
12      }, Promise.resolve(null));
13      return newValues;
14  });
15 }
```

In each version we just chain each asynchronous operation off of the previous one. It's annoying that we need to create a "blank" promise that is simply used to start the chain, but it's a necessary evil. Also, we need to explicitly assign values to the `newValues` array (assuming you want to return those), which is another necessary evil, though maybe not quite as evil. I personally think the version with `forEach` is slightly easier to read thanks to its explicit nature, but it's a stylistic choice and `reduce` works perfectly for this situation.

Conclusion

I used to think promises weren't very straight-forward and even had a hard time finding a reason to use them over standard callbacks, but the more I need them, the more useful I find them to be, but I also find them to be more complicated with numerous ways they can be used, as shown above.

Understanding your options and keeping a list of patterns you can follow greatly helps when the time comes to use them. If you don't already have these patterns embedded in your brain, you may want to save them

somewhere so you have them handy when you need them.

Well, that's all for today. God bless!
Happy Coding!



Author: Joe Zimmerman



Joe Zimmerman has been doing web development ever since he found an HTML

book on his dad's shelf when he was 12. Since then, JavaScript has grown in popularity and he has become passionate about it. He also loves to teach others through his blog and other popular blogs. When he's not writing code, he's spending time with his wife and children and leading them in God's Word.

#JavaScript

Comments

Share

#design patterns #promises

We were unable to load Disqus. If you are a moderator please see our [troubleshooting guide](#).



© 2016 Joe Zimmerman

[Home](#)

[Recommendations](#)

[About](#)

[Contact Me](#)

[Privacy Policy](#)