SUBSCRIBE

Understanding the Node.js Event Loop

by Trevor Norris

SHARE

Node's "event loop" is central to being able to handle high throughput scenarios. It is a magical place filled with unicorns and rainbows, and is the reason Node can essentially be "single threaded" while still allowing an arbitrary number of operations to be handled in the background. This post will shed light on how the event loop operates so you too can enjoy the magic.

Event Driven Programming

The first thing needed in order to understand the event loop is an understanding of the event-driven programming paradigm. This has been well understood since the 1960's. Today, event-driven programming is largely used in UI applications. A major usage of JavaScript is to interact with the DOM, so the use of event-based APIs was natural.

Defined simply: event-driven programming is application flow control that is determined by events or changes in state. The general implementation is to have a central mechanism that listens for events and calls a callback function once

an event has been detected (i.e. state has changed). Sound familiar? It should. That's the basic principle behind Node's event loop.

For those familiar with client-side JavaScript development, think of all the `.on*()` methods, such as `element.onclick()`, that are used in conjunction with DOM Elements to convey user interacti This pattern works well when a single item can emit many possible events. Node uses this pattern in the form of the `EventEmitter`, and is located in places such as `Server`, `Socket` and the `'http'` module. It's useful when we need to emit more than one type of state change from a single instance.

Another common pattern is succeed or fail. There are two common implementations around today. First is the "error back" callback style, where the error of the call is the first argument passed to the callback. The second has emerged with ES6, using `Promises`.

The `'fs'` module mostly uses the error back callback style. It would technically be possible to emit additional events for some calls, such as `fs.readFile()`, but the API was made to only alert the user if the desired operation succeeded or if something failed. Thi selection was an architecture decision and not due to technical limitations.

A common misconception is that event emitters are somehow asynchronous in nature on their own, but this is incorrect. The following is a trivial code snippet to demonstrate this.

```
function MyEmitter() {
  EventEmitter.call(this);
}
util.inherits(MyEmitter, EventEmitter);

MyEmitter.prototype.doStuff = function doStuff() {
  console.log('before')
  emitter.emit('fire')
  console.log('after')}
};

var me = new MyEmitter();
me.on('fire', function() {
  console.log('emit fired');
});

me.doStuff();
// Output:
// before
// emit fired
// after
```

`EventEmitter` often appears asynchronous because it is regularly used to signal the completion of asynchronous operati

but the `EventEmitter` API is
entirely synchronous. The emit function may be called asynchronously, but note that all the listener functions will be executed synchronously, in the order they were added, before any execution can continue in statements following the call to emit.

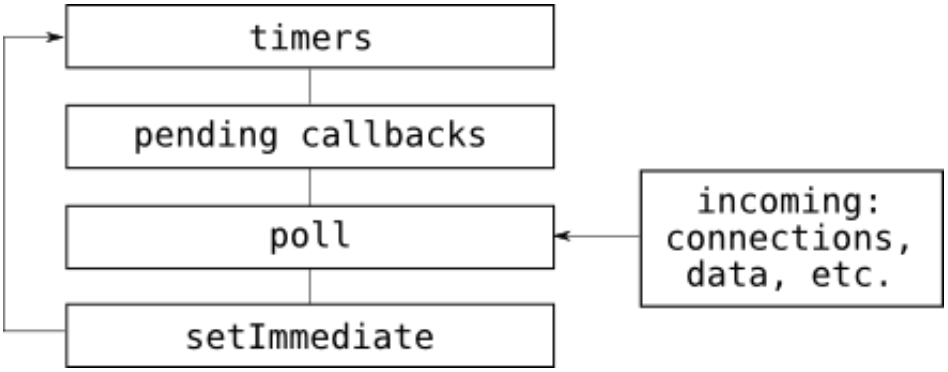
Mechanical Overview

Node itself depends on multiple libraries. One of those is libuv, the magical library that handles the queueing and processing of asynchronous events. For the remainder of this post please keep in mind that I won't distinguish if a point made relates directly to Node or libuv.

Node utilizes as much of what's already available from the operating system's kernel as possible. Responsibilities like making write requests, holding connections and more are therefore delegated to and handled by the system. For example, incoming connections are queued by the system until they can be handled by Node.

You may have heard that Node has a thread pool, and might be wondering "if Node pushes all those responsibilities down why would a thread pool be needed?" It's because the kernel doesn't support doing everything asynchronously. In those cases Node has to lock a thread for the duration of the operation so it can continue executing the event loop without blocking.

Here is a simplified diagram to explain the mechanical overview of when things run:



A couple important notes about the inner workings of the event loop that would be difficult to include in the diagram are:

- All callbacks scheduled via `process.nextTick()` are run at the end of a phase of the event loop (e.g. timers) bef transitioning to the next phase. This creates the potential to unintentionally starve the event loop with recursive to `process.nextTick()`.

- "Pending callbacks" is where callbacks are queued to run that aren't handled by any other phase (e.g. a callback passed to `fs.write()`).

Event Emitter and the Event Loop

To simplify interaction with the event loop the `EventEmitter` was created. It is a generic wrapper that more easily allows creating event-based APIs. Because of some of the confusion that surrounds how these two interact we'll now address common points that tend to trip up developers.

The following example shows how forgetting that emitting events happens synchronously can cause events to be missed by the user.

```
// Post v0.10, require('events').EventEmitter is not necessary.
var EventEmitter = require('events');
var util = require('util');

function MyThing() {
  EventEmitter.call(this);

  doFirstThing();
  this.emit('thing1');
}
util.inherits(MyThing, EventEmitter);
```


The flaw with the above is that `'thing1'` can never be captured by the user because `MyThing()` must finish instantiating before listening for any events. Here is a simple solution that also doesn't require any additional closures:

```
var EventEmitter = require('events');
var util = require('util');

function MyThing() {
  EventEmitter.call(this);

  doFirstThing();
  setImmediate(emitThing1, this);
}
util.inherits(MyThing, EventEmitter);

function emitThing1(self) {
  self.emit('thing1');
}

var mt = new MyThing();

mt.on('thing1', function onThing1() {
  // Whoot!
});
```


The following would also work, but at a drastic performance cost:

```
function MyThing() {
  EventEmitter.call(this);

  doFirstThing();
  // Using Function#bind() makes the world much slower.
  setImmediate(this.emit.bind(this, 'thing1'));
}
util.inherits(MyThing, EventEmitter);
```

Another problem case is with emitting errors. Figuring out problems with your application can be hard enough, but losing the call stack can make it impossible. A call stack is lost when an `Error` is instantiated on the far end of an asynchronous request. The two most reasonable solutions to get around this problem are to emit synchronously or to make sure other important information propagates with the error. The following example shows each one being used:

```
MyThing.prototype.foo = function foo() {
  // This error will be emitted asynchronously.
  var er = doFirstThing();
  if (er) {
    // The error needs to be created immediately to preserve
    // the call stack.
    setImmediate(emitError, this, new Error('Bad stuff'));
    return;
  }

  // Emit the error immediately so it can be handled.
  var er = doSecondThing();
  if (er) {
    this.emit('error', 'More bad stuff');
    return;
  }
}
```

Consider the situation. It may be possible that the error being emitted should be handled immediately, before the application proceeds executing. Or it may be something as trivial as a bad argument that needs to be reported and can easily be handled later. Also, it's not a good idea to have a constructors that emit errors, since the object instance's construction may very well be incomplete. Just throw an exception in that case.

Wrapping Up

This post has been very light on the technical details and inner workings of the event loop. Which was deliberate. That information will be covered in the future, but first we needed to make sure every one was on the same page with these basics. Look forward to another article on how the event loop interacts with your system's kernel to achieve the asynchronous magic that allows Node to run.

Read More

More articles from the NodeSource team

View all Articles

Node.js v5.9.1 Rel

by Jeremiah Senkpiel on 27th of March, 2016

This week's stable fixes a possible crash (a regression performance of Immediate processing. As with all rele drop-in replacements for previous versions. Full Char documentation-only commits. 9 only modify tests an

Read More

Node.js v4.4.1 Rel

by Jeremiah Senkpiel on 27th of March, 2016

This LTS release is the result of a buildup of regular m
note beyond the usual stability improvements and bu
upgrades should be drop-in replacements for previou

Read More

NodeSource Partr
N|Solid as Enterpr
Google Cloud Plat

by Mark Piening on 27th of March, 2016

N|Solid, the enterprise Node.js platform, is the best s
solution for running Node.js applications on Google C
provider of an instrumented Node.js runtime on the C

Read More

COMPANY

About

Press

N|Sight

Blog

OFFERINGS

N|Ship

N|Solid

N|Support

Training

Resources

SOCIAL

Twitter

Github

LinkedIn

Medium