

Controlling Flow

callbacks made are easy

EventEmitters are Easy

- Responding to events is a solved problem
(At least for JavaScripters)
- Very similar to DOM coding
- `thing.on("event", doSomething)`
- easy.

callbacks are hard

- Most common complaint about nodejs:

Ew, callbacks? Ugly nasty nesting indented forever spaghetti code? Just to open a file?!

YOU'VE GOTTA BE FRICKIN KIDDING ME.

- Variant:

Why doesn't this work?

<link to severely broken code>

It's worse than that

- Most **objects** in NodeJS are Event Emitters (http server/client, etc.)
- Most low level **functions** take callbacks. (posix API, DNS lookups, etc.)
- Beyond "hello, world" it gets tricky.

What's actually hard?

- Doing a bunch of things in a specific order.
- Knowing when stuff is done.
- Handling failures.
- Breaking up functionality into parts
(infinitely nested inline callbacks)

Common Mistakes

- Abandoning convention and consistency.
- Putting all callbacks inline.
- Using libraries without grokking them.
- Trying to make async code look sync.*

*controversial

In my opinion, promises are a perfectly fine way to solve this problem. But they're complicated under the hood, and making async code look sync can cause weird expectations. Also, people like to talk about a "Promise" like it's one kind of thing, when it's actually a very general pattern.

flow control libraries

- There are approximately 7 gillion flow control libraries in the node.js ecosystem.
- Everyone likes their own the best.
- Obvious solution: Write your own.
- Let's do that now.

This is a learning exercise

- The goal is not to write the ideal flow control library.
- The goal is simplicity and understanding, with very little magic.
- Please **don't** hold questions for the end.
- Please **do** try this at home.

First Priority: A Really Cool Name

- Be descriptive, but not much to describe.
- So minimal, you can write it in a slide show.
- <http://github.com/isaacs/slide-flow-control>
- `npm install slide`
- Hellz.Yeah.

Define Conventions

- Two kinds of functions:

Actors: Take action

Callbacks: Get results

- Essentially the continuation pattern.
Resulting code **looks** similar to fibers, but is **much** simpler to implement.
- Bonus: node works this way in the lowlevel APIs already, and it's very flexible.

Callbacks

- Simple responders
- Must always be prepared to handle errors!
(That's why it's the first argument.)
- Often inline anonymous, but not always.
- Can trap and call other callbacks with modified data, or to pass errors upwards.

Actors

- Last argument is a callback.
- If any error occurs, and can't be handled, pass it to the callback and return.
- Must not throw. Return value ignored.
- `return x ==> return cb(null, x)`
- `throw er ==> return cb(er)`

Actor Example

```
function actor (some, args, cb) {  
  // last argument is callback  
  // optional args:  
  if (!cb &&  
      typeof(args) === "function")  
    cb = args, args = []  
  
  // do something, and then:  
  
  if (failed) cb(new Error(  
    "failed!"))  
  else cb(null, optionalData)  
}
```

Actor Example

```
// return true if a path is either
// a symlink or a directory.
function isLinkOrDir (path, cb) {
  fs.lstat(path, function (er, s) {
    if (er) return cb(er)
    return cb(null,
      s.isDirectory() ||
      s.isSymbolicLink())
  })
}
```

Actor Example

```
// return true if a path is either
// a symlink or a directory.
function isLinkOrDir (path, cb) {
  fs.lstat(path, function (er, s) {
    if (er) return cb(er)
    return cb(null,
      s.isDirectory() ||
      s.isSymbolicLink())
  })
}
```

Actor Example

```
// return true if a path is either
// a symlink or a directory.
function isLinkOrDir (path, cb) {
  fs.lstat(path, function (er, s) {
    if (er) return cb(er)
    return cb(null,
      s.isDirectory() ||
      s.isSymbolicLink())
  })
}
```


Actor Example

```
// return true if a path is either
// a symlink or a directory.
function isLinkOrDir (path, cb) {
  fs.lstat(path, function (er, s) {
    if (er) return cb(er)
    return cb(null,
      s.isDirectory() ||
      s.isSymbolicLink())
  })
}
```

Actor Composition

```
// return true if a path is either
// a symlink or a directory, and also
// ends in ".bak"
function isLinkDirBak (path, cb) {
  return isLinkOrDir(path,
    function (er, ld) {
      return cb(er, ld &&
        path.substr(-4) === ".bak")
    })
}
```

Actor Composition

```
// return true if a path is either
// a symlink or a directory, and also
// ends in ".bak"
function isLinkDirBak (path, cb) {
  return isLinkOrDir(path,
    function (er, ld) {
      return cb(er, ld &&
        path.substr(-4) === ".bak")
    })
}
```

Actor Composition

```
// return true if a path is either
// a symlink or a directory, and also
// ends in ".bak"
function isLinkDirBak (path, cb) {
  return isLinkOrDir(path,
    function (er, ld) {
      return cb(er, ld &&
        path.substr(-4) === ".bak")
    })
}
```

usecase: asyncMap

- I have a list of 10 files, and need to read all of them, and then continue when they're all done.
- I have a dozen URLs, and need to fetch them all, and then continue when they're all done.
- I have 4 connected users, and need to send a message to all of them, and then continue when that's done.

usecase: asyncMap

- I have a list of n things, and I need to *dosomething* with all of them, in parallel, and get the results once they're all complete.

```
function asyncMap (list, fn, cb_) {  
  var n = list.length  
    , results = []  
    , errState = null  
  function cb (er, data) {  
    if (errState) return  
    if (er) return cb(errState = er)  
    results.push(data)  
    if (-- n === 0)  
      return cb_(null, results)  
  }  
  list.forEach(function (l) {  
    fn(l, cb)  
  })  
}
```

```
function asyncMap (list, fn, cb_) {  
  var n = list.length  
    , results = []  
    , errState = null  
  function cb (er, data) {  
    if (errState) return  
    if (er) return cb(errState = er)  
    results.push(data)  
    if (-- n === 0)  
      return cb_(null, results)  
  }  
  list.forEach(function (l) {  
    fn(l, cb)  
  })  
}
```



```
function asyncMap (list, fn, cb_) {  
  var n = list.length  
    , results = []  
    , errState = null  
  function cb (er, data) {  
    if (errState) return  
    if (er) return cb(errState = er)  
    results.push(data)  
    if (-- n === 0)  
      return cb_(null, results)  
  }  
  list.forEach(function (l) {  
    fn(l, cb)  
  })  
}
```

```
function asyncMap (list, fn, cb_) {  
  var n = list.length  
    , results = []  
    , errState = null  
  function cb (er, data) {  
    if (errState) return  
    if (er) return cb(errState = er)  
    results.push(data)  
    if (-- n === 0)  
      return cb_(null, results)  
  }  
  list.forEach(function (l) {  
    fn(l, cb)  
  })  
}
```

usecase: asyncMap

```
function writeFiles (files, what, cb) {  
  asyncMap( files  
    , function (f, cb) {  
      fs.writeFile(f, what, cb)  
    }  
    , cb  
  )  
}
```

```
writeFiles([my, file, list], "foo", cb)
```

asyncMap

- note that asyncMap itself is an Actor function, so you can asyncMap your asyncMaps, dawg.
- This implementation is fine if order doesn't matter, but what if it does?

asyncMap - ordered

- close over the array index in the generated cb function.
- match up results to their original index.

```
function asyncMap (list, fn, cb_) {  
  var n = list.length  
    , results = []  
    , errState = null  
  function cbGen (i) {  
    return function cb (er, data) {  
      if (errState) return  
      if (er) return cb(errState = er)  
      results[i] = data  
      if (-- n === 0)  
        return cb_(null, results)  
    }  
  }  
  list.forEach(function (l, i) {  
    fn(l, cbGen(i))  
  })  
}
```

```
function asyncMap (list, fn, cb_) {  
  var n = list.length  
    , results = []  
    , errState = null  
  function cbGen (i) {  
    return function cb (er, data) {  
      if (errState) return  
      if (er) return cb(errState = er)  
      results[i] = data  
      if (-- n === 0)  
        return cb_(null, results)  
    }  
  }  
  list.forEach(function (l, i) {  
    fn(l, cbGen(i))  
  })  
}
```

usecase: chain

- I have to do a bunch of things, in order. Get db credentials out of a file, read the data from the db, write that data to another file.
- If anything fails, do not continue.


```
function chain (things, cb) {  
  ;(function LOOP (i, len) {  
    if (i >= len) return cb()  
    things[i](function (er) {  
      if (er) return cb(er)  
      LOOP(i + 1, len)  
    })  
  })(0, things.length)  
}
```

```
function chain (things, cb) {  
  ;(function LOOP (i, len) {  
    if (i >= len) return cb()  
    things[i](function (er) {  
      if (er) return cb(er)  
      LOOP(i + 1, len)  
    })  
  })(0, things.length)  
}
```

```
function chain (things, cb) {  
  ;(function LOOP (i, len) {  
    if (i >= len) return cb()  
    things[i](function (er) {  
      if (er) return cb(er)  
      LOOP(i + 1, len)  
    })  
  })(0, things.length)  
}
```

```
function chain (things, cb) {  
  ;(function LOOP (i, len) {  
    if (i >= len) return cb()  
    things[i](function (er) {  
      if (er) return cb(er)  
      LOOP(i + 1, len)  
    })  
  })(0, things.length)  
}
```

usecase: chain

- Still have to provide an array of functions, which is a lot of boilerplate, and a pita if your functions take args
"function (cb){blah(a,b,c,cb)}"
- Results are discarded, which is a bit lame.
- No way to branch.

reducing boilerplate

- convert an array of [fn, args] to an actor that takes no arguments (except cb)
- A bit like Function#bind, but tailored for our use-case.
- bindActor(obj, "method", a, b, c)
bindActor(fn, a, b, c)
bindActor(obj, fn, a, b, c)

```
function bindActor () {  
    var args =  
        Array.prototype.slice.call  
        (arguments) // jswtf.  
    , obj = null  
    , fn  
    if (typeof args[0] === "object") {  
        obj = args.shift()  
        fn = args.shift()  
        if (typeof fn === "string")  
            fn = obj[ fn ]  
    } else fn = args.shift()  
    return function (cb) {  
        fn.apply(obj, args.concat(cb))  
    }  
}
```

```
function bindActor () {  
    var args =  
        Array.prototype.slice.call  
        (arguments) // jswtf.  
    , obj = null  
    , fn  
    if (typeof args[0] === "object") {  
        obj = args.shift()  
        fn = args.shift()  
        if (typeof fn === "string")  
            fn = obj[ fn ]  
    } else fn = args.shift()  
    return function (cb) {  
        fn.apply(obj, args.concat(cb))  
    }  
}
```



```
function bindActor () {  
    var args =  
        Array.prototype.slice.call  
        (arguments) // jswtf.  
    , obj = null  
    , fn  
    if (typeof args[0] === "object") {  
        obj = args.shift()  
        fn = args.shift()  
        if (typeof fn === "string")  
            fn = obj[ fn ]  
    } else fn = args.shift()  
    return function (cb) {  
        fn.apply(obj, args.concat(cb))  
    }  
}
```

```
function bindActor () {  
    var args =  
        Array.prototype.slice.call  
        (arguments) // jswtf.  
    , obj = null  
    , fn  
    if (typeof args[0] === "object") {  
        obj = args.shift()  
        fn = args.shift()  
        if (typeof fn === "string")  
            fn = obj[ fn ]  
    } else fn = args.shift()  
    return function (cb) {  
        fn.apply(obj, args.concat(cb))  
    }  
}
```

bindActor

- Some obvious areas for improvement.
- They wouldn't fit on a slide.
- Left as an exercise for the reader.

```
function chain (things, cb) {  
  ;(function LOOP (i, len) {  
    if (i >= len) return cb()  
    if (Array.isArray(things[i]))  
      things[i] = bindActor.apply  
        (null, things[i])  
    things[i](function (er) {  
      if (er) return cb(er)  
      LOOP(i + 1, len)  
    })  
  })(0, things.length)  
}
```

```
function chain (things, cb) {  
  ;(function LOOP (i, len) {  
    if (i >= len) return cb()  
    if (Array.isArray(things[i]))  
      things[i] = bindActor.apply  
        (null, things[i])  
    things[i](function (er) {  
      if (er) return cb(er)  
      LOOP(i + 1, len)  
    })  
  })(0, things.length)  
}
```

chain: branching

- Skip over falsey arguments
- ```
chain([doThing && [thing, a, b, c]
 , isFoo && [doFoo, "foo"]
 , subChain &&
 [chain, [one, two]]
], cb)
```

```
function chain (things, cb) {
 ;(function LOOP (i, len) {
 if (i >= len) return cb()
 if (Array.isArray(things[i]))
 things[i] = bindActor.apply
 (null, things[i])
 if (!things[i])
 return LOOP(i + 1, len)
 things[i](function (er) {
 if (er) return cb(er)
 LOOP(i + 1, len)
 })
 })(0, things.length)
}
```





```

chain.first = {} ; chain.last = {}
function chain (things, res, cb) {
 if (!cb) cb = res , res = []
 ;(function LOOP (i, len) {
 if (i >= len) return cb(null, res)
 if (Array.isArray(things[i]))
 things[i] = bindActor.apply(null,
 things[i].map(function(i) {
 return (i===chain.first) ? res[0]
 : (i===chain.last)
 ? res[res.length - 1] : i })))
 if (!things[i]) return LOOP(i + 1, len)
 things[i](function (er, data) {
 res.push(er || data)
 if (er) return cb(er, res)
 LOOP(i + 1, len)
 })
 })(0, things.length) }

```

```
chain.first = {} ; chain.last = {}
function chain (things, res, cb) {
 if (!cb) cb = res , res = []
 ;(function LOOP (i, len) {
 if (i >= len) return cb(null, res)
 if (Array.isArray(things[i]))
 things[i] = bindActor.apply(null,
 things[i].map(function(i) {
 return (i===chain.first) ? res[0]
 : (i===chain.last)
 ? res[res.length - 1] : i })))
 if (!things[i]) return LOOP(i + 1, len)
 things[i](function (er, data) {
 res.push(er || data)
 if (er) return cb(er, res)
 LOOP(i + 1, len)
 })
 })(0, things.length) }
```

```
chain.first = {} ; chain.last = {}
function chain (things, res, cb) {
 if (!cb) cb = res , res = []
 ;(function LOOP (i, len) {
 if (i >= len) return cb(null, res)
 if (Array.isArray(things[i]))
 things[i] = bindActor.apply(null,
 things[i].map(function(i) {
 return (i===chain.first) ? res[0]
 : (i===chain.last)
 ? res[res.length - 1] : i })))
 if (!things[i]) return LOOP(i + 1, len)
 things[i](function (er, data) {
 res.push(er || data)
 if (er) return cb(er, res)
 LOOP(i + 1, len)
 })
 })(0, things.length) }
```

```
chain.first = {} ; chain.last = {}
function chain (things, res, cb) {
 if (!cb) cb = res , res = []
 ;(function LOOP (i, len) {
 if (i >= len) return cb(null, res)
 if (Array.isArray(things[i]))
 things[i] = bindActor.apply(null,
 things[i].map(function(i) {
 return (i===chain.first) ? res[0]
 : (i===chain.last)
 ? res[res.length - 1] : i })))
 if (!things[i]) return LOOP(i + 1, len)
 things[i](function (er, data) {
 res.push(er || data)
 if (er) return cb(er, res)
 LOOP(i + 1, len)
 })
 })(0, things.length) }
```

```
chain.first = {} ; chain.last = {}
function chain (things, res, cb) {
 if (!cb) cb = res , res = []
 ;(function LOOP (i, len) {
 if (i >= len) return cb(null, res)
 if (Array.isArray(things[i]))
 things[i] = bindActor.apply(null,
 things[i].map(function(i) {
 return (i===chain.first) ? res[0]
 : (i===chain.last)
 ? res[res.length - 1] : i })))
 if (!things[i]) return LOOP(i + 1, len)
 things[i](function (er, data) {
 res.push(er || data)
 if (er) return cb(er, res)
 LOOP(i + 1, len)
 })
 })(0, things.length) }
```

```

chain.first = {} ; chain.last = {}
function chain (things, res, cb) {
 if (!cb) cb = res , res = []
 ;(function LOOP (i, len) {
 if (i >= len) return cb(null, res)
 if (Array.isArray(things[i]))
 things[i] = bindActor.apply(null,
 things[i].map(function(i) {
 return (i===chain.first) ? res[0]
 : (i===chain.last)
 ? res[res.length - 1] : i })))
 if (!things[i]) return LOOP(i + 1, len)
 things[i](function (er, data) {
 res.push(er || data)
 if (er) return cb(er, res)
 LOOP(i + 1, len)
 })
 })(0, things.length) }

```

Ok, this can't get any  
bigger or it won't fit.

# Non-trivial Use Case

- Read number files in a directory
- Add the results together
- Ping a web service with the result
- Write the response to a file
- Delete the number files

```
var chain = require("./chain.js")
 , asyncMap = require("./async-map.js")
function myProgram (cb) {
 var res = [], last = chain.last
 , first = chain.first
 chain
 ([[fs, "readdir", "the-directory"]
 , [readFiles, "the-directory", last]
 , [sum, last]
 , [ping, "POST", "example.com", 80
 , "/foo", last]
 , [fs, "writeFile", "result.txt", last]
 , [rmFiles, "./the-directory", first]
]
 , res
 , cb
)
}
```



```
var chain = require("./chain.js")
 , asyncMap = require("./async-map.js")
function myProgram (cb) {
 var res = [], last = chain.last
 , first = chain.first
 chain
 ([[fs, "readdir", "the-directory"]
 , [readFiles, "the-directory", last]
 , [sum, last]
 , [ping, "POST", "example.com", 80
 , "/foo", last]
 , [fs, "writeFile", "result.txt", last]
 , [rmFiles, "./the-directory", first]
]
 , res
 , cb
)
}
```

```
var chain = require("./chain.js")
 , asyncMap = require("./async-map.js")
function myProgram (cb) {
 var res = [], last = chain.last
 , first = chain.first
 chain
 ([[fs, "readdir", "the-directory"]
 , [readFiles, "the-directory", last]
 , [sum, last]
 , [ping, "POST", "example.com", 80
 , "/foo", last]
 , [fs, "writeFile", "result.txt", last]
 , [rmFiles, "./the-directory", first]
]
 , res
 , cb
)
}
```

```
var chain = require("./chain.js")
 , asyncMap = require("./async-map.js")
function myProgram (cb) {
 var res = [], last = chain.last
 , first = chain.first
 chain
 ([[fs, "readdir", "the-directory"]
 , [readFiles, "the-directory", last]
 , [sum, last]
 , [ping, "POST", "example.com", 80
 , "/foo", last]
 , [fs, "writeFile", "result.txt", last]
 , [rmFiles, "./the-directory", first]
]
 , res
 , cb
)
}
```

```
var chain = require("./chain.js")
 , asyncMap = require("./async-map.js")
function myProgram (cb) {
 var res = [], last = chain.last
 , first = chain.first
 chain
 ([[fs, "readdir", "the-directory"]
 , [readFiles, "the-directory", last]
 , [sum, last]
 , [ping, "POST", "example.com", 80
 , "/foo", last]
 , [fs, "writeFile", "result.txt", last]
 , [rmFiles, "./the-directory", first]
]
 , res
 , cb
)
}
```

```
var chain = require("./chain.js")
 , asyncMap = require("./async-map.js")
function myProgram (cb) {
 var res = [], last = chain.last
 , first = chain.first
 chain
 ([[fs, "readdir", "the-directory"]
 , [readFiles, "the-directory", last]
 , [sum, last]
 , [ping, "POST", "example.com", 80
 , "/foo", last]
 , [fs, "writeFile", "result.txt", last]
 , [rmFiles, "./the-directory", first]
]
 , res
 , cb
)
}
```

```
var chain = require("./chain.js")
 , asyncMap = require("./async-map.js")
function myProgram (cb) {
 var res = [], last = chain.last
 , first = chain.first
 chain
 ([[fs, "readdir", "the-directory"]
 , [readFiles, "the-directory", last]
 , [sum, last]
 , [ping, "POST", "example.com", 80
 , "/foo", last]
 , [fs, "writeFile", "result.txt", last]
 , [rmFiles, "./the-directory", first]
]
 , res
 , cb
)
}
```

# Convention Profits

- Consistent API from top to bottom.
- Sneak in at any point to inject functionality. (testable, reusable, etc.)
- When ruby and python users whine, you can smile condescendingly.