ASYNC / AWAIT

HOW TO AVOID CALLBACK HELL?

VIEW ON GITHUB

Raymond Sze

WHAT IS THUNK?

```
function sumThunk(number1, number2, callback) {
  setImmediate(() => {
    try {
      const sum = parseInt(number1) + parseInt(number2);
      callback(null, sum);
    } catch (error) {
      callback(error);
    }
  });
}
```

- Thunk can accepts any number of arguments
- The last argument must be callback function
- Callback function accepts 2 arguments (error, result)

THUNK EXAMPLE

```
import fs from 'fs';
import mongoose from 'mongoose';
const Schema = mongoose.Schema;
const Person = mongoose.model('Person', new Schema({ username: String }));

// fs.readFile is Thunk function
fs.readFile('/testFile', 'utf8', (err, data) => {
  if (err) throw err;
  console.log(data);
});

// Person.findOne is Thunk function
Person.findOne({ 'username': 'Ghost' }, (err, person) => {
  if (err) throw err;
  console.log(`username is ${person.username}`);
})
```

- NodeJs api design are using thunk
- Lots of library are using thunk (e.g mongoose)
- But... Thunk makes Callback Hell!

Callback Hell

NESTING CALLS

```
// 4 times nested calls...
fs.readFile('/test1', 'utf-8', (err1, data1) => {
  if (err1) handleError(err1);
  fs.readFile(data1, 'utf-8', (err2, data2) => {
    if (err2) handleError(err2);
    fs.readFile(data2, 'utf-8', (err3, data3) => {
      if (err3) handleError(err3);
      fs.readFile(data3, 'utf-8', (err4, data4) => {
        if (err4) handleError(err4);
        doSomething(data4);
      });
    });
  });
});
```

PARALLEL ASYNC PROCESSING

```
// parallel processing
const completed = [];
let error = null;
fs.readFile('/test1', 'utf-8', (err, data) => {
  if (err && !error) error = err;
  else completed.push(data);
});
fs.readFile('/test2', 'utf-8', (err, data) => {
  if (err && !error) error = err;
  else completed.push(data);
});

while(completed.length === 2 || error) {
  ....
}
```

NOT EXCEPTION SAFE

```
try {
  fs.readFile('/test1', 'utf-8', (err, data) => {
    throw new Error('not exception safe');
  });
} catch (e) {
  // the 'not exception safe' error never go here...
  console.log('caughtException');
  console.error(e);
}

process.on('uncaughtException', (err) => {
  // the 'not exception safe' error go here...
  console.log('uncaughtException');
  console.error(err);
});
```

NO CENTRALIZED ERROR HANDLING

```
// 4 handleError(...)
// actually they are same error handler...
fs.readFile('/test1', 'utf-8', (err1, data1) => {
  if (err1) handleError(err1);
  fs.readFile(data1, 'utf-8', (err2, data2) => {
    if (err2) handleError(err2);
    fs.readFile(data2, 'utf-8', (err3, data3) => {
      if (err3) handleError(err3);
      fs.readFile(data3, 'utf-8', (err4, data4) => {
        if (err4) handleError(err4);
        doSomething(data4);
      });
    });
  });
});
```

WHAT WE WANT TO SOLVE?

- Easy to read, No Long Nesting Callback
- Easy Parallel Async
- Particular Error Handling
- Centralized Error Handling
- Exception Safe

Solution1: Async.js

ASYNC.JS

```
async.waterfall([
  function(callback) {
    fs.readFile('/test1', 'utf-8', callback);
  },
  function(data1, callback) {
    fs.readFile(data1, 'utf-8', callback);
  },
  function(data2, callback) {
    fs.readFile(data2, 'utf-8', callback);
  },
  function(data3, callback) {
    fs.readFile(data3, 'utf-8', callback);
  }
], function (err, data4) {
  if (err) handleError(err);
  doSomething(data4);
});
```

ASYNC.JS

```
async.parallel([
  function(callback) {
    fs.readFile('/test1', 'utf-8', callback);
  },
  function(callback) {
    fs.readFile('/test2', 'utf-8', callback);
  },
], function (err, results) {
  if (err) handleError(err);
  doSomething(results);
});
```

ASYNC.JS

- Need external library async.js
- Group callbacks to Array
- Still not Good to Read
- Still not Exception Safe

Solution2: Promise

WHAT IS PROMISE

```
// Thunk version
function sumThunk(number1, number2, callback) {
  setImmediate(() => {
    try {
      const sum = parseInt(number1) + parseInt(number2);
      callback(null, sum);
    } catch (error) {
      callback(error);
    }
  });
}

// Promise version
function sumAsync(number1, number2) {
  return new Promise((resolve, reject) => {
    try {
      const sum = parseInt(number1) + parseInt(number2);
      resolve(sum);
    } catch (error) {
      reject(error); // use throw error is also acceptable
    }
  });
}
```

PROMISE API

```
// Promise.prototype.then, how to get the result
sumAsync(1, 2).then(
  data => {
    // now data is 1 + 2 = 3
    // ...
  }
);
// Promise.prototype.catch, how to catch the error
sumAsync(1, 'a').catch(
  error => {
    // now the error is about unable to parse 'a'
    // ...
  }
);
// promise is chain-able
sumAsync(1, 2)
  .then(data => {
    // ...
  })
  .catch(error => {
    // ...
  });
```

PROMISE API

```
// promise is chain-able
sumAsync(1, 2)
  .then(data =>
    // now data is 1 + 2 = 3
    sumAsync(data, 3)
  )
  // you can add .catch here to catch the error from sumAsync(1, 2)
  .then(data =>
    // now data is 3 + 3 = 6
    sumAsync(data, 5) // 6 + 5 = 11;
  )
  .catch(error => {
    // here error is from previous then
    // ...
  });

// promise is compose-able
// here sumAsync(3,4) won't wait sumAsync(1,2) complete
var p = Promise.all([sumAsync(1,2), sumAsync(3,4)]);
p.then(data =>
  // now data is [3, 7];
  data[0] + data[1];
);
```

Imagine if fs.readFile return Promise

NESTING CALLS

```
// 4 times nested calls...
fs.readFile('/test1', 'utf-8', (err1, data1) => {
  if (err1) throw err1;
  fs.readFile(data1, 'utf-8', (err2, data2) => {
    if (err2) throw err2;
    fs.readFile(data2, 'utf-8', (err3, data3) => {
      if (err3) throw err3;
      fs.readFile(data3, 'utf-8', (err4, data4) => {
        if (err4) throw err4;
        doSomething(data4);
      });
    });
  });
});
// Promise version
fs.readFile('/test1', 'utf-8')
  .then(data1 =>
    fs.readFile(data1, 'utf-8')
  ).then(data2 =>
    fs.readFile(data2, 'utf-8')
  ).then(data3 =>
    fs.readFile(data3, 'utf-8')
  ).then(data4 => {
    doSomething(data4);
  }).catch(error => {
    throw error; // centralized error handlers
  });
```

PARALLEL ASYNC

```
// parallel processing
const completed = [];
let error = null;
fs.readFile('/test1', 'utf-8', (err, data) => {
  if (err && !error) error = err;
  else completed.push(data);
});
fs.readFile('/test2', 'utf-8', (err, data) => {
  if (err && !error) error = err;
  else completed.push(data);
});

while(completed.length === 2 || error) {
  ....
}

// Promise version
Promise.all([
  fs.readFile('/test1', 'utf-8'),
  fs.readFile('/test2', 'utf-8'),
]).then(...);
```

EXCPETION SAFE

```
try {
  fs.readFile('/test1', 'utf-8', (err, data) => {
    throw new Error('not exception safe');
  });
} catch (e) {
  // the 'not exception safe' error never go here...
  console.log('caughtException');
  console.error(e);
}

// Promise version
fs.readFile('/test1', 'utf-8').then(data => {
  throw new Error('not exception safe');
}).catch(e => {
  console.log('caughtException');
  console.error(e);
});
```

But fs.readFile NOT return Promise...

THUNK TO PROMISE

```
fs.readFileAsync = function (...args) {
  return new Promise((resolve, reject) => {
    fs.readFile.apply(this, [...args, (error, data) => {
      if (error) reject(error);
      else resolve(data);
    }]);
  });
};

// The following have same effect to the above
import Bluebird from 'bluebird';
fs.readFileAsync = Bluebird.promisify(fs.readFile);

// Or shorthand to convert all function to promise
const fsAsync = Bluebird.promisifyAll(fs);
```

- Thunk to Promise is easy
- You don't need write yourself
- Checkout Bluebird.promisfy and Bluebird.promisifyAll

PROMISE

- Exception Safe
- Easy to Parallel Async
- Still not Good to Read, then vs Array
- Need convert Thunk to Promise
- 2 way to throw Exception (By throw or reject)

Solution3: Coroutine (Promise + Generator)

WHAT IS GENERATOR

```
// function* mean it is a generator function
function* gen() {
  // yield is special syntax for generator function
  yield 1;
  var v = yield 2;
  var f = yield 3 + v;
  return f + 4;
}

var g = gen();
g.next(); // { value: 1, done: false };
g.next(); // { value: 2, done: false };
g.next(4); // { value: 7, done: false };
g.next(5); // { value: 9, done: true };
```

- Simulate the 'wait' effect
- Array, String, Map, Set are using Generator as Iterator
- Accessible by [Symbol.iterator]

How Generator Help?

- How about the generator could auto 'next'?
- How about yielding Promise?

COROUTINE

```
function coroutine(g) {
  const gen = g();
  const next = gen.next();
  const nextStep = (p) => {
    if (p.done) return p.value;
    return p.value.then((v) => {
      return nextStep(gen.next(v));
    });
  };
  return nextStep(next);
}
```

- Assume value must be Promise object
- Auto pass the previous value to call .next
- Now yield become 'wait' if wrapped by coroutine

Assume fs.readFile return Promise again...

COROUTINE

```
// thunk version
fs.readFile('/test1', 'utf-8', (err1, data1) => {
  if (err1) handleError(err1);
  fs.readFile(data1, 'utf-8', (err2, data2) => {
    if (err2) handleError(err2);
    fs.readFile(data2, 'utf-8', (err3, data3) => {
      if (err3) handleError(err3);
      fs.readFile(data3, 'utf-8', (err4, data4) => {
        if (err4) handleError(err4);
        doSomething(data4);
      });
    });
  });
});

// coroutine version
coroutine(function *() {
  const data1 = yield fs.readFile('/test1');
  const data2 = yield fs.readFile(data1);
  const data3 = yield fs.readFile(data2);
  const data4 = yield fs.readFile(data3);
  doSomething(data4);
}).catch(err => {
  handleError(err);
});
```

PROBLEM SOLVED?

```
// No Nesting Calls
coroutine(function *() {
  // Parallel Async
  const [data1, data2] = yield Promise.all([
    fs.readFile('/test1'),
    fs.readFile('/test2'),
  ]);
  let data3;
  try {
    data3 = yield fs.readFile(data1 + data2);
  } catch (e) {
    // Particular Error Handling
    handleError(e);
  }
  const data4 = yield fs.readFile(data3);
  doSomething(data4);
}).catch(err => {
  // Centralized Error Handling
  handleError(err);
});
```

- Need write coroutine function
- Still need convert Thunk to Promise

CO.JS, BLUEBIRD.JS

```
// bluebird
import Bluebird from 'bluebird';
const fsAsync = Bluebird.promisifyAll(fs);
const func = Bluebird.coroutine(function *() {
  const data1 = yield fsAsync.readFile('/test1');
  const data2 = yield fsAsync.readFile(data1);
  doSomething(data2);
});
func().then(...).catch(...);
// co
import co from 'co';
const func = co.wrap(function *() {
  const data1 = yield fs.readFile('/test1');
  const data2 = yield fs.readFile(data1);
  doSomething(data2);
});
func().then(...).catch(...);
```

- Bluebird.coroutine support Promise only
- co.wrap support Promise, Thunk, Generator, Constant

EVERYTHING TO PROMISE

```
// Constant
Promise.resolve(1);
Promise.resolve('a');
Promise.resolve(['a', 2]);
Promise.resolve({ a: 20, b: 30 });
// Thunk
function sample(...args) {
  return new Promise((resolve, reject) => {
    thunk.apply(this, ...args, (err, ...args2) => {
      if (err) reject(err);
      else resolve(...args2);
    });
  });
}
// Generator
co.wrap(generator);
```

COROUTINE

- Exception Safe
- Easy to Parallel Async by Promise.all
- Make use of Generator syntax, Easy to Read
- With co.js, no need convert Thunk to Promise

ASYNC, AWAIT

```
// No Nesting Calls
async function demo() {
  // Parallel Async
  const [data1, data2] = await Promise.all([
    fs.readFile('/test1'),
    fs.readFile('/test2'),
  ]);
  let data3;
  try {
    data3 = await fs.readFile(data1 + data2);
  } catch (e) {
    // Particular Error Handling
    handleError(e);
  }
  const data4 = await fs.readFile(data3);
  doSomething(data4);
}).catch(err => {
  // Centralized Error Handling
  handleError(err);
});
```

- async = coroutine, await = yield
- support Promise only

ASYNC, AWAIT

- Exception Safe
- Easy to Parallel Async by Promise.all
- async, await is ES2015 standard (Node7 support)
- Need convert Thunk to Promise (Bluebird could help you)

What if non Node 7 environment?

BABEL, ESLINT

```
// .babelrc
{
  "presets": ["latest", "stage-0"],
}
// .eslintrc
{
  "extends": "airbnb",
  "env": {
    "node": true
  }
}
// package.json
{
  ...
  "devDependencies": {
    "babel-cli": "^6.18.0",
    "babel-preset-latest": "^6.16.0",
    "babel-preset-stage-0": "^6.16.0",
    "eslint": "^3.9.1",
    "eslint-config-airbnb": "^12.0.0",
    "eslint-plugin-import": "^1.16.0",
    "eslint-plugin-jsx-a11y": "^2.2.3",
    "eslint-plugin-react": "^6.5.0"
    ...
  }
}
```

RECOMMENDATION

- Write your function returning Promise, no more thunk
- Use async and await syntax as first preference
- Use Bluebird.coroutine as second preference
- Use Promise.all to handle parallel tasks
- Use Bluebird.promisify or Bluebird.promisifyAll to convert thunk to promise
- Review your application if Exception Safe is handled correctly