A Tale of Two (or more) Asynchrony

by Adam Crabtree
Founder of BayNode, Noderiety

Asynchrony

Definition:

- *any* object or function that represents an asynchronous value or error.

Asynchrony: Representations

Definition:

- ***any* object or function** that represents an asynchronous value or error.

Asynchrony: Examples

```
// Callbacks
function callback(err, value) {
  if(err) {
    // do something with err
    return;
  }
  // do something with value
}

// Promises
promise.then(function(value) {
    // do something with value
  }
, function(err) {
    // do something with err
  })
```

Asynchrony: Contracts

- All asynchrony have a contract:
  - Errbacks (Implicit): https://gist.github.com/CrabDude/10907185
  - Promises A+ (Explicit via a spec): http://promises-aplus.github.io/promises-spec/

- Ideally, they have tests as well: https://github.com/promises-aplus/promises-tests

Asynchrony: Errback Contract

1. Function that takes 2 arguments
   - first argument is an error
   - second argument is the result
   - Never pass both
   - error should be instanceof Error
2. Must never excecute on the same tick of the event loop
3. Must be passed as last argument to function
4. Return value is ignored
5. Must not throw / must pass resulting errors
6. Must never be called more than once

Asynchrony: Promises Spec

Promises/A+

An open standard for sound, interoperable JavaScript promises—by implementers, for implementers.

Terminology

"promise" is an object or function with a then method whose behavior conforms to this specification.
"thenable" is an object or function that defines a then method.
"value" is any legal JavaScript value (including undefined, a thenable, or a promise).
"exception" is a value that is thrown using the throw statement.
"reason" is a value that indicates why a promise was rejected.

Requirements

Promise States

A promise must be in one of three states: pending, fulfilled, or rejected.
When pending, a promise:
may transition to either the fulfilled or rejected state.
When fulfilled, a promise:
must not transition to any other state.
must have a value, which must not change.
When rejected, a promise:
must not transition to any other state.
must have a reason, which must not change.
Here, "must not change" means immutable identity (i.e. ===), but does not imply deep immutability.

The then Method

A promise must provide a then method to access its current or eventual value or reason.
A promise's then method accepts two arguments:
promise.then(onFulfilled, onRejected)
Both onFulfilled and onRejected are optional arguments:
If onFulfilled is not a function, it must be ignored.
If onRejected is not a function, it must be ignored.
If onFulfilled is a function:
it must be called after promise is fulfilled, with promise's value as its first argument.
it must not be called before promise is fulfilled.
it must not be called more than once.
If onRejected is a function,
it must be called after promise is rejected, with promise's reason as its first argument.
it must not be called before promise is rejected.
it must not be called more than once.
onFulfilled or onRejected must not be called until the execution context stack contains only platform code. [3.1].
onFulfilled and onRejected must be called as functions (i.e. with no this value). [3.2]
then may be called multiple times on the same promise.
If/when promise is fulfilled, all respective onFulfilled callbacks must execute in the order of their originating calls to then.
If/when promise is rejected, all respective onRejected callbacks must execute in the order of their originating calls to then.
then must return a promise [3.3]. promise2 = promise1.then(onFulfilled, onRejected);

If either onFulfilled or onRejected returns a value x, run the Promise Resolution Procedure[[Resolve]](promise2, x).
If either onFulfilled or onRejected throws an exception e, promise2 must be rejected with e as the reason.
If onFulfilled is not a function and promise1 is fulfilled, promise2 must be fulfilled with the same value as promise1.
If onRejected is not a function and promise1 is rejected, promise2 must be rejected with the same reason as promise1.

The Promise Resolution Procedure

The **promise resolution procedure** is an abstract operation taking as input a promise and a value, which we denote as [[Resolve]](promise, x). If x is a thenable, it attempts to make promise adopt the state of x, under the assumption that x behaves at least somewhat like a promise. Otherwise, it fulfills promise with the value x.
This treatment of thenables allows promise implementations to interoperate, as long as they expose a Promises/A+-compliant then method. It also allows Promises/A+ implementations to "assimilate" nonconformant implementations with reasonable then methods.
To run [[Resolve]](promise, x), perform the following steps:
If promise and x refer to the same object, reject promise with a TypeError as the reason.
If x is a promise, adopt its state [3.4]:
If x is pending, promise must remain pending until x is fulfilled or rejected.
If/when x is fulfilled, fulfill promise with the same value.
If/when x is rejected, reject promise with the same reason.
Otherwise, if x is an object or function,
Let then be x.then. [3.5]
If retrieving the property x.then results in a thrown exception e, reject promise with e as the reason.
If then is a function, call it with x as this, first argument resolvePromise, and second argumentrejectPromise, where:
If/when resolvePromise is called with a value y, run [[Resolve]](promise, y).
If/when rejectPromise is called with a reason r, reject promise with r.
If both resolvePromise and rejectPromise are called, or multiple calls to the same argument are made, the first call takes precedence, and any further calls are ignored.
If calling then throws an exception e,

- If resolvePromise or rejectPromise have been called, ignore it.
- Otherwise, reject promise with e as the reason.

If then is not a function, fulfill promise with x.
If x is not an object or function, fulfill promise with x.
If a promise is resolved with a thenable that participates in a circular thenable chain, such that the recursive nature of [[Resolve]](promise, thenable) eventually causes [[Resolve]](promise, thenable) to be called again, following the above algorithm will lead to infinite recursion. Implementations are encouraged, but not required, to detect such recursion and reject promise with an informative TypeError as the reason. [3.6]

Asynchrony: Results

Definition:

- *any* object or function that represents **an asynchronous value or error.**

Asynchrony

- There are two primary results to any operation, a value or an error.
- Asynchrony must deal with both.
  - Sometimes manually
  - Sometimes automatically

Asynchrony: Bad Error Handling

- **Throwing the error violates the callback contract:**

```
function readCurrentFileAsString(callback) {
	fs.readFile(__filename, function(err, data) {
		// BAD
		if (err) throw err

		callback(null, String(data))
	})
}
```

Asynchrony: Good Error Handling

- **Bubble the error to maintain the callback contract:**

```
function readCurrentFileAsString(callback) {
	fs.readFile(__filename, function(err, data) {
		callback(err, data && String(data))
	})
}

function readCurrentFileAsString() {
   return fs.promisifiedReadFile(__filename)
        .then(String)
}
```

Control-flow: Why?

- Juggle values, enforce ordering, coalesce errors

```
function readCurrentFileAsString(callback) {
    stepup([
        function($) {
            fs.readFile(__filename, $.first())
            setTimeout($.none(), 1000)
        },
        function($, data) {
            return String(data)
        }
    ], callback)
}
```

Control-flow: Why? (Promises)

- Juggle values, enforce ordering, coalesce errors

```
function readCurrentFileAsString(callback) {
   var deferred = q.defer();
   setTimeout(deferred.resolve, 1000)
   return q.all([
          fs.promisifiedReadFile(__filename),
          deferred.promise
       ])
       .then(String)
}
```

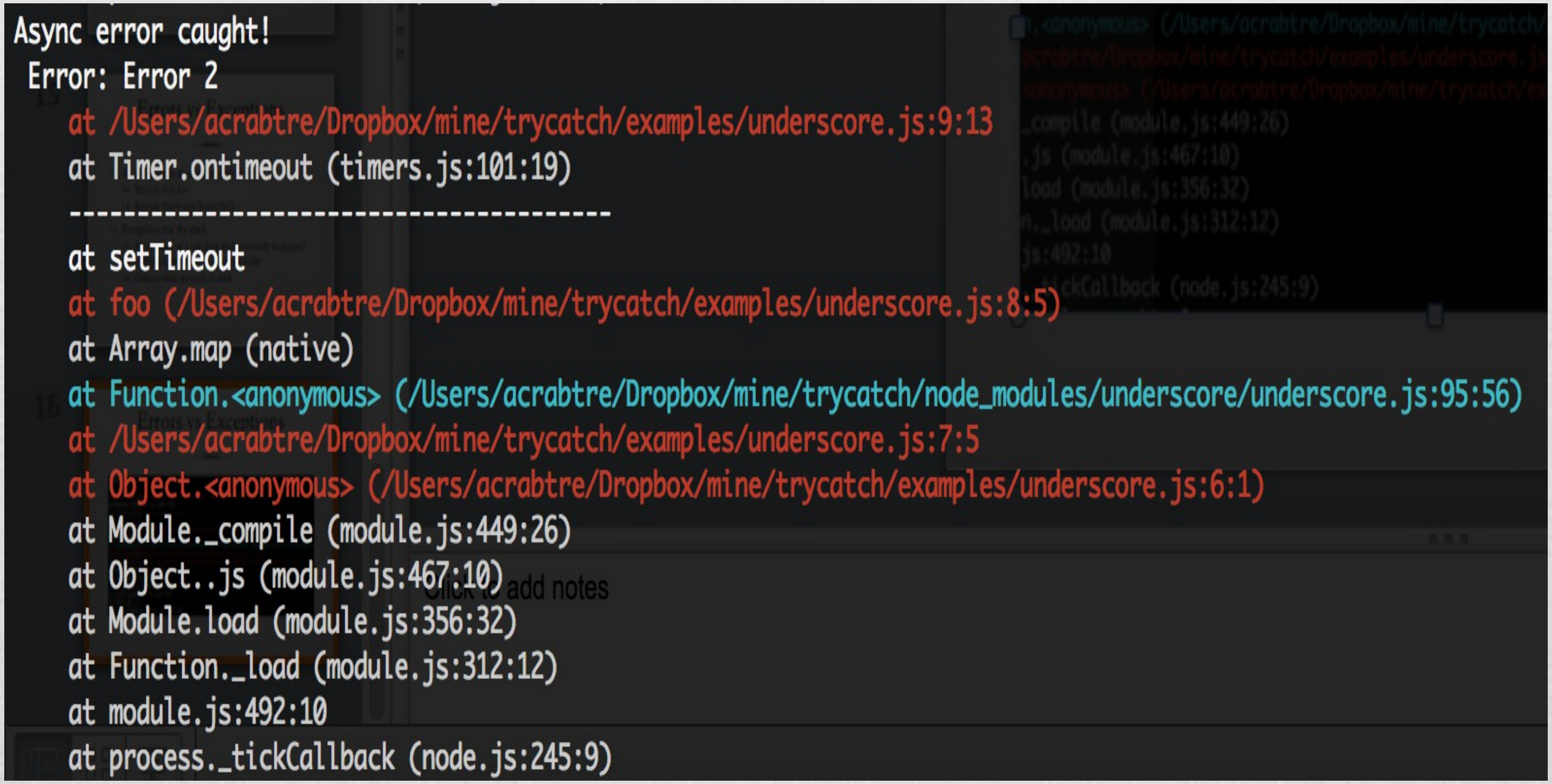
Control-flow: Guardians of Contract

- Control-flow is the gatekeeper that enforces the asynchrony contract
- When calling into 3rd party code, don’t trust it to properly follow the callback contract
- Ensure it doesn’t throw an uncaughtException
  - with trycatch (built into stepup)
  - Promises wrap all callbacks in try/catch

Errors vs Exceptions

- An exception is an unrecoverable error
  - Implicit: null.foo
  - Explicit: throw new Error('fail')
- Exceptions tear the stack
  - Every frame on a call stack must eventually be popped
  - Ultimately, the cause of unrecoverability
  - Desired when we need to crash

Errors vs Exceptions



```
Async error caught!
 Error: Error 2
    at /Users/acrabtre/Dropbox/mine/trycatch/examples/underscore.js:9:13
    at Timer.ontimeout (timers.js:101:19)
    ----------------------------------------
    at setTimeout
    at foo (/Users/acrabtre/Dropbox/mine/trycatch/examples/underscore.js:8:5)
    at Array.map (native)
    at Function.<anonymous> (/Users/acrabtre/Dropbox/mine/trycatch/node_modules/underscore/underscore.js:95:56)
    at /Users/acrabtre/Dropbox/mine/trycatch/examples/underscore.js:7:5
    at Object.<anonymous> (/Users/acrabtre/Dropbox/mine/trycatch/examples/underscore.js:6:1)
    at Module._compile (module.js:449:26)
    at Object..js (module.js:467:10)
    at Module.load (module.js:356:32)
    at Function._load (module.js:312:12)
    at module.js:492:10
    at process._tickCallback (node.js:245:9)
```

Exceptions in Node.js

- Exceptions are communicated via process.on(‘uncaughtException’)

trycatch

- trycatch, asynchronous try/catch support: https://github.com/CrabDude/trycatch
- Wraps core/userland boundary in try/catch
  - Prevents core stack from tearing, ensuring core stack frames are always allowed to unwind

```
trycatch(function() {
	setTimeout(function() {throw new Error(v)}, 10)
}, function(err) {
	console.log("Async error caught!\n", err.stack);
});
```