Async.js

Async is a utility module which provides straight-forward, powerful functions for working with asynchronous JavaScript.

Async provides around 70 functions that include the usual 'functional' suspects (`map`, `reduce`, `filter`, `each`…) as well as some common patterns for asynchronous control flow (`parallel`, `series`, `waterfall`…). All these functions assume you follow the Node.js convention of providing a single callback as the last argument of your asynchronous function -- a callback which expects an Error as its first argument -- and calling the callback once.

Quick Examples

```
async.map(['file1','file2','file3'], fs.stat, function(err, results){
    // results is now an array of stats for each file
});

async.filter(['file1','file2','file3'], function(filePath, callback) {
  fs.access(filePath, function(err) {
    callback(null, !err)
  });
}, function(err, results){
    // results now equals an array of the existing files
});

async.parallel([
    function(){ ... },
    function(){ ... }
], callback);

async.series([
    function(){ ... },
    function(){ ... }
]);
```

There are many more functions available so take a look at the docs below for a full list. This module aims to be comprehensive, so if you feel anything is missing please create a GitHub issue for it.

Common Pitfalls

Synchronous iteration functions

If you get an error like `RangeError: Maximum call stack size exceeded.` or other stack overflow issues when using async, you are likely using a synchronous iteratee. By *synchronous* we mean a function that calls its callback on the same tick in the javascript event loop, without doing any I/O or using any timers. Calling many callbacks iteratively will quickly overflow the stack. If you run into this issue, just defer your callback with `async.setImmediate` to start a new call stack on the next tick of the event loop.

This can also arise by accident if you callback early in certain cases:

```
async.eachSeries(hugeArray, function iteratee(item, callback) {
    if (inCache(item)) {
        callback(null, cache[item]); // if many items are cached, you'll overflow
    } else {
        doSomeIO(item, callback);
    }
}, function done() {
    //...
});
```

Just change it to:

```
async.eachSeries(hugeArray, function iteratee(item, callback) {
    if (inCache(item)) {
        async.setImmediate(function () {
            callback(null, cache[item]);
        });
    } else {
        doSomeIO(item, callback);
        //...
    }
});
```

Async does not guard against synchronous iteratees for performance reasons. If you are still running into stack overflows, you can defer as suggested above, or wrap functions with `async.ensureAsync` Functions that are asynchronous by their nature do not have this problem and don't need the extra callback deferral.

If JavaScript's event loop is still a bit nebulous, check out this article or this talk for more detailed information about how it works.

Multiple callbacks

Make sure to always `return` when calling a callback early, otherwise you will cause multiple callbacks and unpredictable behavior in many cases.

```
async.waterfall([
    function (callback) {
        getSomething(options, function (err, result) {
            if (err) {
                callback(new Error("failed getting something:" + err.message));
                // we should return here
            }
            // since we did not return, this callback still will be called and
            // `processData` will be called twice
            callback(null, result);
        });
    },
    processData
], done)
```

It is always good practice to `return callback(err, result)` whenever a callback call is not the last statement of a function.

Documentation

Some functions are also available in the following forms:

- `<name>Series` - the same as `<name>` but runs only a single async operation at a time
- `<name>Limit` - the same as `<name>` but runs a maximum of `limit` async operations at a time

Collections

- `each`, `eachSeries`, `eachLimit`
- `forEachOf`, `forEachOfSeries`, `forEachOfLimit`
- `map`, `mapSeries`, `mapLimit`
- `filter`, `filterSeries`, `filterLimit`
- `reject`, `rejectSeries`, `rejectLimit`
- `reduce`, `reduceRight`
- `detect`, `detectSeries`, `detectLimit`
- `sortBy`
- `some`, `someLimit`, `someSeries`
- `every`, `everyLimit`, `everySeries`
- `concat`, `concatSeries`

Control Flow

- `series`
- `parallel`, `parallelLimit`
- `whilst`, `doWhilst`
- `until`, `doUntil`
- `during`, `doDuring`
- `forever`
- `waterfall`
- `compose`
- `seq`
- `applyEach`, `applyEachSeries`
- `queue`, `priorityQueue`
- `cargo`
- `auto`
- `autoInject`
- `retry`
- `retryable`
- `iterator`
- `times`, `timesSeries`, `timesLimit`
- `race`

Utils

- `apply`
- `nextTick`
- `memoize`
- `unmemoize`
- `ensureAsync`
- `constant`
- `asyncify`
- `wrapSync`
- `log`
- `dir`
- `noConflict`
- `timeout`

Collections

Collection methods can iterate over Arrays, Objects, Maps, Sets, and any object that implements the ES2015 iterator protocol.

each(coll, iteratee, [callback])

Applies the function `iteratee` to each item in `coll`, in parallel.
The `iteratee` is called with an item from the list, and a callback for when it has finished. If the `iteratee` passes an error to its `callback`, the main `callback` (for the `each` function) is immediately called with the error.

Note, that since this function applies `iteratee` to each item in parallel, there is no guarantee that the iteratee functions will complete in order.

Arguments

- `coll` - A collection to iterate over.
- `iteratee(item, callback)` - A function to apply to each item in `coll`. The iteratee is passed a `callback(err)` which must be called once it has completed. If no error has occurred, the `callback` should be run without arguments or with an explicit `null` argument. The array index is not passed to the iteratee. If you need the index, use `forEachOf`.
- `callback(err)` - *Optional* A callback which is called when all `iteratee` functions have finished, or an error occurs.

Examples

```
// assuming openFiles is an array of file names and saveFile is a function
// to save the modified contents of that file:

async.each(openFiles, saveFile, function(err){
    // if any of the saves produced an error, err would equal that error
});
```

```
// assuming openFiles is an array of file names

async.each(openFiles, function(file, callback) {

  // Perform operation on file here.
  console.log('Processing file ' + file);

  if( file.length > 32 ) {
    console.log('This file name is too long');
    callback('File name too long');
  } else {
    // Do work to process file here
    console.log('File processed');
    callback();
  }
}, function(err){
    // if any of the file processing produced an error, err would equal that error
    if( err ) {
      // One of the iterations produced an error.
      // All processing will now stop.
      console.log('A file failed to process');
    } else {
      console.log('All files have been processed successfully');
    }
});
```

Related

- eachSeries(coll, iteratee, [callback])
- eachLimit(coll, limit, iteratee, [callback])

forEachOf(coll, iteratee, [callback])

Like `each`, except that it passes the key (or index) as the second argument to the iteratee.

Arguments

- `coll` - A collection to iterate over.
- `iteratee(item, key, callback)` - A function to apply to each item in `coll`. The `key` is the item's key, or index in the case of an array. The iteratee is passed a `callback(err)` which must be called once it has completed. If no error has occurred, the callback should be run without arguments or with an explicit `null` argument.
- `callback(err)` - *Optional* A callback which is called when all `iteratee` functions have finished, or an error occurs.

Example

```
var obj = {dev: "/dev.json", test: "/test.json", prod: "/prod.json"};
var configs = {};

async.forEachOf(obj, function (value, key, callback) {
    fs.readFile(__dirname + value, "utf8", function (err, data) {
        if (err) return callback(err);
        try {
            configs[key] = JSON.parse(data);
        } catch (e) {
            return callback(e);
        }
        callback();
    });
}, function (err) {
    if (err) console.error(err.message);
    // configs is now a map of JSON data
    doSomethingWith(configs);
})
```

Related

- forEachOfSeries(coll, iteratee, [callback])
- forEachOfLimit(coll, limit, iteratee, [callback])

map(coll, iteratee, [callback])

Produces a new collection of values by mapping each value in `coll` through
the `iteratee` function. The `iteratee` is called with an item from `coll` and a
callback for when it has finished processing. Each of these callback takes 2 arguments:
an `error`, and the transformed item from `coll`. If `iteratee` passes an error to its
callback, the main `callback` (for the `map` function) is immediately called with the error.

Note, that since this function applies the `iteratee` to each item in parallel,
there is no guarantee that the `iteratee` functions will complete in order.
However, the results array will be in the same order as the original `coll`.

Arguments

- `coll` - A collection to iterate over.
- `iteratee(item, callback)` - A function to apply to each item in `coll`. The iteratee is passed a `callback(err, transformed)` which must be called once it has completed with an error (which can be `null`) and a transformed item.
- `callback(err, results)` - *Optional* A callback which is called when all `iteratee` functions have finished, or an error occurs. Results is an array of the transformed items from the `coll`.

Example

```
async.map(['file1','file2','file3'], fs.stat, function(err, results){
    // results is now an array of stats for each file
});
```

Related

- mapSeries(coll, iteratee, [callback])
- mapLimit(coll, limit, iteratee, [callback])

filter(coll, iteratee, [callback])

Alias: `select`

Returns a new array of all the values in `coll` which pass an async truth test.
This operation is performed in parallel,
but the results array will be in the same order as the original.

Arguments

- `coll` - A collection to iterate over.
- `iteratee(item, callback)` - A truth test to apply to each item in `coll`. The `iteratee` is passed a `callback(err, truthValue)`, which must be

called with a boolean argument once it has completed.
- `callback(err, results)` - *Optional* A callback which is called after all the `iteratee` functions have finished.

Example

```
async.filter(['file1','file2','file3'], function(filePath, callback) {
  fs.access(filePath, function(err) {
    callback(null, !err)
  });
}, function(err, results){
    // results now equals an array of the existing files
});
```

Related

- filterSeries(coll, iteratee, [callback])
- filterLimit(coll, limit, iteratee, [callback])

reject(coll, iteratee, [callback])

The opposite of `filter`. Removes values that pass an `async` truth test.

Related

- rejectSeries(coll, iteratee, [callback])
- rejectLimit(coll, limit, iteratee, [callback])

reduce(coll, memo, iteratee, [callback])

Aliases: `inject`, `foldl`

Reduces `coll` into a single value using an async `iteratee` to return
each successive step. `memo` is the initial state of the reduction.
This function only operates in series.

For performance reasons, it may make sense to split a call to this function into
a parallel map, and then use the normal `Array.prototype.reduce` on the results.
This function is for situations where each step in the reduction needs to be async;
if you can get the data before reducing it, then it's probably a good idea to do so.

Arguments

- `coll` - A collection to iterate over.
- `memo` - The initial state of the reduction.
- `iteratee(memo, item, callback)` - A function applied to each item in the array to produce the next step in the reduction. The `iteratee` is passed a `callback(err, reduction)` which accepts an optional error as its first argument, and the state of the reduction as the second. If an error is passed to the callback, the reduction is stopped and the main `callback` is immediately called with the error.
- `callback(err, result)` - *Optional* A callback which is called after all the `iteratee` functions have finished. Result is the reduced value.

Example

```
async.reduce([1,2,3], 0, function(memo, item, callback){
    // pointless async:
    process.nextTick(function(){
        callback(null, memo + item)
    });
}, function(err, result){
    // result is now equal to the last value of memo, which is 6
});
```

reduceRight(coll, memo, iteratee, [callback])

Alias: `foldr`

Same as `reduce`, only operates on `coll` in reverse order.

---

detect(coll, iteratee, [callback])

Alias: `find`

Returns the first value in `coll` that passes an async truth test. The `iteratee` is applied in parallel, meaning the first iteratee to return `true` will fire the detect `callback` with that result. That means the result might not be the first item in the original `coll` (in terms of order) that passes the test.

If order within the original `coll` is important, then look at `detectSeries`.

Arguments

- `coll` - A collection to iterate over.
- `iteratee(item, callback)` - A truth test to apply to each item in `coll`. The iteratee is passed a `callback(err, truthValue)` which must be called with a boolean argument once it has completed.
- `callback(err, result)` - *Optional* A callback which is called as soon as any iteratee returns `true`, or after all the `iteratee` functions have finished. Result will be the first item in the array that passes the truth test (iteratee) or the value `undefined` if none passed.

Example

```
async.detect(['file1','file2','file3'], function(filePath, callback) {
  fs.access(filePath, function(err) {
    callback(null, !err)
  });
}, function(err, result){
    // result now equals the first file in the list that exists
});
```

Related

- detectSeries(coll, iteratee, [callback])
- detectLimit(coll, limit, iteratee, [callback])

---

sortBy(coll, iteratee, [callback])

Sorts a list by the results of running each `coll` value through an async `iteratee`.

Arguments

- `coll` - A collection to iterate over.
- `iteratee(item, callback)` - A function to apply to each item in `coll`. The iteratee is passed a `callback(err, sortValue)` which must be called once it has completed with an error (which can be `null`) and a value to use as the sort criteria.
- `callback(err, results)` - *Optional* A callback which is called after all the `iteratee` functions have finished, or an error occurs. Results is the items from the original `coll` sorted by the values returned by the `iteratee` calls.

Example

```
async.sortBy(['file1','file2','file3'], function(file, callback){
    fs.stat(file, function(err, stats){
        callback(err, stats.mtime);
    });
}, function(err, results){
    // results is now the original array of files sorted by
    // modified date
});
```

Sort Order

By modifying the callback parameter the sorting order can be influenced:

```
//ascending order
async.sortBy([1,9,3,5], function(x, callback){
    callback(null, x);
}, function(err,result){
    //result callback
} );

//descending order
async.sortBy([1,9,3,5], function(x, callback){
    callback(null, x*-1);    //<- x*-1 instead of x, turns the order around
}, function(err,result){
    //result callback
} );
```

---

some(coll, iteratee, [callback])

Alias: `any`

Returns `true` if at least one element in the `coll` satisfies an async test.
If any iteratee call returns `true`, the main `callback` is immediately called.

Arguments

- `coll` - A collection to iterate over.
- `iteratee(item, callback)` - A truth test to apply to each item in the array in parallel. The iteratee is passed a `callback(err, truthValue)` which must be called with a boolean argument once it has completed.
- `callback(err, result)` - *Optional* A callback which is called as soon as any iteratee returns `true`, or after all the iteratee functions have finished. Result will be either `true` or `false` depending on the values of the async tests.

Example

```
async.some(['file1','file2','file3'], function(filePath, callback) {
  fs.access(filePath, function(err) {
    callback(null, !err)
  });
}, function(err, result){
    // if result is true then at least one of the files exists
});
```

Related

- someSeries(coll, iteratee, callback)
- someLimit(coll, limit, iteratee, callback)

---

every(coll, iteratee, [callback])

Alias: `all`

Returns `true` if every element in `coll` satisfies an async test.
If any iteratee call returns `false`, the main `callback` is immediately called.

Arguments

- `coll` - A collection to iterate over.
- `iteratee(item, callback)` - A truth test to apply to each item in the collection in parallel. The iteratee is passed a `callback(err, truthValue)` which must be called with a boolean argument once it has completed.
- `callback(err, result)` - *Optional* A callback which is called after all the `iteratee` functions have finished. Result will be either `true` or `false` depending on the values of the async tests.

Example

```
async.every(['file1','file2','file3'], function(filePath, callback) {
  fs.access(filePath, function(err) {
    callback(null, !err)
  });
}, function(err, result){
    // if result is true then every file exists
});
```

Related

- everySeries(coll, iteratee, callback)
- everyLimit(coll, limit, iteratee, callback)

---

concat(coll, iteratee, [callback])

Applies `iteratee` to each item in `coll`, concatenating the results. Returns the concatenated list. The `iteratee`s are called in parallel, and the results are concatenated as they return. There is no guarantee that the results array will be returned in the original order of `coll` passed to the `iteratee` function.

Arguments

- `coll` - A collection to iterate over.
- `iteratee(item, callback)` - A function to apply to each item in `coll`. The iteratee is passed a `callback(err, results)` which must be called once it has completed with an error (which can be `null`) and an array of results.
- `callback(err, results)` - *Optional* A callback which is called after all the `iteratee` functions have finished, or an error occurs. Results is an array containing the concatenated results of the `iteratee` function.

Example

```
async.concat(['dir1','dir2','dir3'], fs.readdir, function(err, files){
    // files is now a list of filenames that exist in the 3 directories
});
```

Related

- concatSeries(coll, iteratee, [callback])

Control Flow

series(tasks, [callback])

Run the functions in the `tasks` collection in series, each one running once the previous function has completed. If any functions in the series pass an error to its callback, no more functions are run, and `callback` is immediately called with the value of the error. Otherwise, `callback` receives an array of results when `tasks` have completed.

It is also possible to use an object instead of an array. Each property will be run as a function, and the results will be passed to the final `callback` as an object instead of an array. This can be a more readable way of handling results from `series`.

Note that while many implementations preserve the order of object properties, the ECMAScript Language Specification explicitly states that

> The mechanics and order of enumerating the properties is not specified.

So if you rely on the order in which your series of functions are executed, and want this to work on all platforms, consider using an array.

Arguments

- `tasks` - A collection containing functions to run, each function is passed a `callback(err, result)` it must call on completion with an error `err` (which can be `null`) and an optional `result` value.
- `callback(err, results)` - An optional callback to run once all the functions have completed. This function gets a results array (or object) containing all the result arguments passed to the `task` callbacks.

Example

```
async.series([
    function(callback){
        // do some stuff ...
        callback(null, 'one');
    },
    function(callback){
        // do some more stuff ...
        callback(null, 'two');
    }
],
// optional callback
function(err, results){
    // results is now equal to ['one', 'two']
});


// an example using an object instead of an array
async.series({
    one: function(callback){
        setTimeout(function(){
            callback(null, 1);
        }, 200);
    },
    two: function(callback){
        setTimeout(function(){
            callback(null, 2);
        }, 100);
    }
},
function(err, results) {
    // results is now equal to: {one: 1, two: 2}
});
```

---

parallel(tasks, [callback])

Run the `tasks` collection of functions in parallel, without waiting until the previous function has completed. If any of the functions pass an error to its callback, the main `callback` is immediately called with the value of the error. Once the `tasks` have completed, the results are passed to the final `callback` as an array.

Note: `parallel` is about kicking-off I/O tasks in parallel, not about parallel execution of code. If your tasks do not use any timers or perform any I/O, they will actually be executed in series. Any synchronous setup sections for each task will happen one after the other. JavaScript remains single-threaded.

It is also possible to use an object instead of an array. Each property will be run as a function and the results will be passed to the final `callback` as an object instead of an array. This can be a more readable way of handling results from `parallel`.

Arguments

- `tasks` - A collection containing functions to run. Each function is passed a `callback(err, result)` which it must call on completion with an error `err` (which can be `null`) and an optional `result` value.
- `callback(err, results)` - An optional callback to run once all the functions have completed successfully. This function gets a results array (or object) containing all the result arguments passed to the task callbacks.

Example

```
async.parallel([
    function(callback){
        setTimeout(function(){
            callback(null, 'one');
        }, 200);
    },
    function(callback){
        setTimeout(function(){
            callback(null, 'two');
        }, 100);
    }
],
// optional callback
function(err, results){
    // the results array will equal ['one','two'] even though
    // the second function had a shorter timeout.
});


// an example using an object instead of an array
async.parallel({
    one: function(callback){
        setTimeout(function(){
            callback(null, 1);
        }, 200);
    },
    two: function(callback){
        setTimeout(function(){
            callback(null, 2);
        }, 100);
    }
},
function(err, results) {
    // results is now equals to: {one: 1, two: 2}
});
```

Related

- parallelLimit(tasks, limit, [callback])

---

whilst(test, fn, callback)

Repeatedly call `fn`, while `test` returns `true`. Calls `callback` when stopped,
or an error occurs.

Arguments

- `test()` - synchronous truth test to perform before each execution of `fn`.
- `fn(callback)` - A function which is called each time `test` passes. The function is passed a `callback(err)`, which must be called once it has completed with an optional `err` argument.
- `callback(err, [results])` - A callback which is called after the test function has failed and repeated execution of `fn` has stopped. `callback` will be passed an error and any arguments passed to the final `fn`'s callback.

Example

```
var count = 0;
```

```
async.whilst(
    function () { return count < 5; },
    function (callback) {
        count++;
        setTimeout(function () {
            callback(null, count);
        }, 1000);
    },
    function (err, n) {
        // 5 seconds have passed, n = 5
    }
);
```

doWhilst(fn, test, callback)

The post-check version of `whilst`. To reflect the difference in
the order of operations, the arguments `test` and `fn` are switched.

`doWhilst` is to `whilst` as `do while` is to `while` in plain JavaScript.

until(test, fn, callback)

Repeatedly call `fn` until `test` returns `true`. Calls `callback` when stopped,
or an error occurs. `callback` will be passed an error and any arguments passed
to the final `fn`'s callback.

The inverse of `whilst`.

doUntil(fn, test, callback)

Like `doWhilst`, except the `test` is inverted. Note the argument ordering differs from `until`.

during(test, fn, callback)

Like `whilst`, except the `test` is an asynchronous function that is passed a callback in the form of `function (err, truth)`. If error is passed to `test` or `fn`, the main callback is immediately called with the value of the error.

Example

```
var count = 0;

async.during(
    function (callback) {
      return callback(null, count < 5);
    },
    function (callback) {
        count++;
        setTimeout(callback, 1000);
    },
    function (err) {
        // 5 seconds have passed
    }
);
```

doDuring(fn, test, callback)

The post-check version of `during`. To reflect the difference in
the order of operations, the arguments `test` and `fn` are switched.

Also a version of `doWhilst` with asynchronous `test` function.

---

forever(fn, [errback])

Calls the asynchronous function `fn` with a callback parameter that allows it to call itself again, in series, indefinitely.

If an error is passed to the callback then `errback` is called with the error, and execution stops, otherwise it will never be called.

```
async.forever(
    function(next) {
        // next is suitable for passing to things that need a callback(err [, whatever]);
        // it will result in this function being called again.
    },
    function(err) {
        // if next is called with a value in its first parameter, it will appear
        // in here as 'err', and execution will stop.
    }
);
```

---

waterfall(tasks, [callback])

Runs the `tasks` array of functions in series, each passing their results to the next in the array. However, if any of the `tasks` pass an error to their own callback, the next function is not executed, and the main `callback` is immediately called with the error.

Arguments

- `tasks` - An array of functions to run, each function is passed a `callback(err, result1, result2, ...)` it must call on completion. The first argument is an error (which can be `null`) and any further arguments will be passed as arguments in order to the next task.
- `callback(err, [results])` - An optional callback to run once all the functions have completed. This will be passed the results of the last task's callback.

Example

```
async.waterfall([
    function(callback) {
        callback(null, 'one', 'two');
    },
    function(arg1, arg2, callback) {
      // arg1 now equals 'one' and arg2 now equals 'two'
        callback(null, 'three');
    },
    function(arg1, callback) {
        // arg1 now equals 'three'
        callback(null, 'done');
    }
], function (err, result) {
    // result now equals 'done'
});
```

Or, with named functions:

```
async.waterfall([
    myFirstFunction,
    mySecondFunction,
    myLastFunction,
], function (err, result) {
    // result now equals 'done'
});
```

```
function myFirstFunction(callback) {
    callback(null, 'one', 'two');
}
function mySecondFunction(arg1, arg2, callback) {
    // arg1 now equals 'one' and arg2 now equals 'two'
    callback(null, 'three');
}
function myLastFunction(arg1, callback) {
    // arg1 now equals 'three'
    callback(null, 'done');
}
```

Or, if you need to pass any argument to the first function:

```
async.waterfall([
    async.apply(myFirstFunction, 'zero'),
    mySecondFunction,
    myLastFunction,
], function (err, result) {
    // result now equals 'done'
});
function myFirstFunction(arg1, callback) {
    // arg1 now equals 'zero'
    callback(null, 'one', 'two');
}
function mySecondFunction(arg1, arg2, callback) {
    // arg1 now equals 'one' and arg2 now equals 'two'
    callback(null, 'three');
}
function myLastFunction(arg1, callback) {
    // arg1 now equals 'three'
    callback(null, 'done');
}
```

compose(fn1, fn2...)

Creates a function which is a composition of the passed asynchronous functions. Each function consumes the return value of the function that follows. Composing functions `f()`, `g()`, and `h()` would produce the result of `f(g(h()))`, only this version uses callbacks to obtain the return values.

Each function is executed with the `this` binding of the composed function.

Arguments

- `functions...` - the asynchronous functions to compose

Example

```
function add1(n, callback) {
    setTimeout(function () {
        callback(null, n + 1);
    }, 10);
}

function mul3(n, callback) {
    setTimeout(function () {
        callback(null, n * 3);
    }, 10);
}

var add1mul3 = async.compose(mul3, add1);

add1mul3(4, function (err, result) {
    // result now equals 15
});
```

seq(fn1, fn2...)

Version of the compose function that is more natural to read.
Each function consumes the return value of the previous function.
It is the equivalent of `compose` with the arguments reversed.

Each function is executed with the `this` binding of the composed function.

Arguments

- `functions...` - the asynchronous functions to compose

Example

```
// Requires lodash (or underscore), express3 and dresende's orm2.
// Part of an app, that fetches cats of the logged user.
// This example uses `seq` function to avoid overnesting and error
// handling clutter.
app.get('/cats', function(request, response) {
    var User = request.models.User;
    async.seq(
        _.bind(User.get, User),  // 'User.get' has signature (id, callback(err, data))
        function(user, fn) {
            user.getCats(fn);      // 'getCats' has signature (callback(err, data))
        }
    )(req.session.user_id, function (err, cats) {
        if (err) {
            console.error(err);
            response.json({ status: 'error', message: err.message });
        } else {
            response.json({ status: 'ok', message: 'Cats found', data: cats });
        }
    });
});
```

applyEach(fns, args..., callback)

Applies the provided arguments to each function in the array, calling
`callback` after all functions have completed. If you only provide the first
argument, then it will return a function which lets you pass in the
arguments as if it were a single function call.

Arguments

- `fns` - the asynchronous functions to all call with the same arguments
- `args...` - any number of separate arguments to pass to the function
- `callback` - the final argument should be the callback, called when all functions have completed processing

Example

```
async.applyEach([enableSearch, updateSchema], 'bucket', callback);

// partial application example:
async.each(
    buckets,
    async.applyEach([enableSearch, updateSchema]),
    callback
);
```

Related

- applyEachSeries(tasks, args..., [callback])

---

queue(worker, [concurrency])

Creates a `queue` object with the specified `concurrency`. Tasks added to the `queue` are processed in parallel (up to the `concurrency` limit). If all `worker`s are in progress, the task is queued until one becomes available. Once a `worker` completes a `task`, that `task`'s callback is called.

Arguments

- `worker(task, callback)` - An asynchronous function for processing a queued task, which must call its `callback(err)` argument when finished, with an optional `error` as an argument. If you want to handle errors from an individual task, pass a callback to `q.push()`.
- `concurrency` - An `integer` for determining how many `worker` functions should be run in parallel. If omitted, the concurrency defaults to `1`. If the concurrency is `0`, an error is thrown.

Queue objects

The `queue` object returned by this function has the following properties and methods:

- `length()` - a function returning the number of items waiting to be processed.
- `started` - a function returning whether or not any items have been pushed and processed by the queue
- `running()` - a function returning the number of items currently being processed.
- `workersList()` - a function returning the array of items currently being processed.
- `idle()` - a function returning false if there are items waiting or being processed, or true if not.
- `concurrency` - an integer for determining how many `worker` functions should be run in parallel. This property can be changed after a `queue` is created to alter the concurrency on-the-fly.
- `push(task, [callback])` - add a new task to the `queue`. Calls `callback` once the `worker` has finished processing the task. Instead of a single task, a `tasks` array can be submitted. The respective callback is used for every task in the list.
- `unshift(task, [callback])` - add a new task to the front of the `queue`.
- `saturated` - a callback that is called when the number of running workers hits the `concurrency` limit, and further tasks will be queued.
- `unsaturated` - a callback that is called when the number of running workers is less than the `concurrency` & `buffer` limits, and further tasks will not be queued.
- `buffer` A minimum threshold buffer in order to say that the `queue` is `unsaturated`.
- `empty` - a callback that is called when the last item from the `queue` is given to a `worker`.
- `drain` - a callback that is called when the last item from the `queue` has returned from the `worker`.
- `paused` - a boolean for determining whether the queue is in a paused state
- `pause()` - a function that pauses the processing of tasks until `resume()` is called.
- `resume()` - a function that resumes the processing of queued tasks when the queue is paused.
- `kill()` - a function that removes the `drain` callback and empties remaining tasks from the queue forcing it to go idle.

Example

```
// create a queue object with concurrency 2

var q = async.queue(function (task, callback) {
    console.log('hello ' + task.name);
    callback();
}, 2);


// assign a callback
q.drain = function() {
    console.log('all items have been processed');
}

// add some items to the queue

q.push({name: 'foo'}, function (err) {
    console.log('finished processing foo');
});
q.push({name: 'bar'}, function (err) {
    console.log('finished processing bar');
});

// add some items to the queue (batch-wise)

q.push([{name: 'baz'},{name: 'bay'},{name: 'bax'}], function (err) {
    console.log('finished processing item');
});

// add some items to the front of the queue

q.unshift({name: 'bar'}, function (err) {
    console.log('finished processing bar');
});
```

priorityQueue(worker, concurrency)

The same as `queue` only tasks are assigned a priority and completed in ascending priority order. There are two differences between `queue` and `priorityQueue` objects:

- `push(task, priority, [callback])` - `priority` should be a number. If an array of `tasks` is given, all tasks will be assigned the same priority.
- The `unshift` method was removed.

cargo(worker, [payload])

Creates a `cargo` object with the specified payload. Tasks added to the cargo will be processed altogether (up to the `payload` limit). If the `worker` is in progress, the task is queued until it becomes available. Once the `worker` has completed some tasks, each callback of those tasks is called. Check out these animations for how `cargo` and `queue` work.

While queue passes only one task to one of a group of workers at a time, cargo passes an array of tasks to a single worker, repeating when the worker is finished.

Arguments

- `worker(tasks, callback)` - An asynchronous function for processing an array of queued tasks, which must call its `callback(err)` argument when finished, with an optional `err` argument.
- `payload` - An optional `integer` for determining how many tasks should be processed per round; if omitted, the default is unlimited.

Cargo objects

The `cargo` object returned by this function has the following properties and methods:

- `length()` - A function returning the number of items waiting to be processed.
- `payload` - An `integer` for determining how many tasks should be process per round. This property can be changed after a `cargo` is created to alter the payload on-the-fly.
- `push(task, [callback])` - Adds `task` to the `queue`. The callback is called once the `worker` has finished processing the task. Instead of a single task, an array of `tasks` can be submitted. The respective callback is used for every task in the list.
- `saturated` - A callback that is called when the `queue.length()` hits the concurrency and further tasks will be queued.
- `empty` - A callback that is called when the last item from the `queue` is given to a `worker`.
- `drain` - A callback that is called when the last item from the `queue` has returned from the `worker`.
- `idle()`, `pause()`, `resume()`, `kill()` - cargo inherits all of the same methods and event callbacks as `queue`

Example

```
// create a cargo object with payload 2

var cargo = async.cargo(function (tasks, callback) {
    for(var i=0; i<tasks.length; i++){
      console.log('hello ' + tasks[i].name);
    }
    callback();
}, 2);


// add some items

cargo.push({name: 'foo'}, function (err) {
    console.log('finished processing foo');
});
cargo.push({name: 'bar'}, function (err) {
    console.log('finished processing bar');
});
cargo.push({name: 'baz'}, function (err) {
    console.log('finished processing baz');
});
```

auto(tasks, [concurrency], [callback])

Determines the best order for running the functions in `tasks`, based on their requirements. Each function can optionally depend on other functions being completed first, and each function is run as soon as its requirements are satisfied.

If any of the functions pass an error to their callback, the `auto` sequence will stop. Further tasks will not execute (so any other functions depending on it will not run), and the main `callback` is immediately called with the error.

Functions also receive an object containing the results of functions which have completed so far as the first argument, if they have dependencies. If a task function has no dependencies, it will only be passed a callback.

```
async.auto({
  // this function will just be passed a callback
  readData: async.apply(fs.readFile, 'data.txt', 'utf-8')
  showData: ['readData', function (results, cb) {
    // results.readData is the file's contents
    // ...
  }]
}, callback);
```

Arguments

- `tasks` - An object. Each of its properties is either a function or an array of requirements, with the function itself the last item in the array. The object's key of a property serves as the name of the task defined by that property, i.e. can be used when specifying requirements for other tasks. The function receives one or two arguments:
    - a `results` object, containing the results of the previously executed functions, only passed if the task has any dependencies,
    - a `callback(err, result)` function, which must be called when finished, passing an `error` (which can be `null`) and the result of the function's execution.
- `concurrency` - An optional `integer` for determining the maximum number of tasks that can be run in parallel. By default, as many as possible.
- `callback(err, results)` - An optional callback which is called when all the tasks have been completed. It receives the `err` argument if any `tasks` pass an error to their callback. Results are always returned; however, if an error occurs, no further `tasks` will be performed, and the results object will only contain partial results.

Example

```
async.auto({
    get_data: function(callback){
        console.log('in get_data');
        // async code to get some data
        callback(null, 'data', 'converted to array');
    },
    make_folder: function(callback){
        console.log('in make_folder');
        // async code to create a directory to store a file in
        // this is run at the same time as getting the data
        callback(null, 'folder');
    },
    write_file: ['get_data', 'make_folder', function(results, callback){
        console.log('in write_file', JSON.stringify(results));
        // once there is some data and the directory exists,
        // write the data to a file in the directory
        callback(null, 'filename');
    }],
    email_link: ['write_file', function(results, callback){
        console.log('in email_link', JSON.stringify(results));
        // once the file is written let's email a link to it...
        // results.write_file contains the filename returned by write_file.
        callback(null, {'file':results.write_file, 'email':'user@example.com'});
    }]
}, function(err, results) {
    console.log('err = ', err);
    console.log('results = ', results);
});
```

This is a fairly trivial example, but to do this using the basic parallel and series functions would look like this:

```
async.parallel([
    function(callback){
        console.log('in get_data');
        // async code to get some data
        callback(null, 'data', 'converted to array');
    },
    function(callback){
        console.log('in make_folder');
        // async code to create a directory to store a file in
        // this is run at the same time as getting the data
        callback(null, 'folder');
    }
],
function(err, results){
    async.series([
        function(callback){
            console.log('in write_file', JSON.stringify(results));
            // once there is some data and the directory exists,
            // write the data to a file in the directory
            results.push('filename');
            callback(null);
        },
        function(callback){
            console.log('in email_link', JSON.stringify(results));
            // once the file is written let's email a link to it...
            callback(null, {'file':results.pop(), 'email':'user@example.com'});
        }
    ]);
});
```

For a complicated series of `async` tasks, using the `auto` function makes adding new tasks much easier (and the code more readable).

---

autoInject(tasks, [callback])

A dependency-injected version of the `auto` function. Dependent tasks are specified as parameters to the function, after the usual callback parameter, with the parameter names matching the names of the tasks it depends on. This can provide even more readable task graphs which can be easier to maintain.

If a final callback is specified, the task results are similarly injected, specified as named parameters after the initial error parameter.

The autoInject function is purely syntactic sugar and its semantics are otherwise equivalent to `auto`.

Arguments

- `tasks` - An object, each of whose properties is a function of the form 'func([dependencies...], callback). The object's key of a property serves as the name of the task defined by that property, i.e. can be used when specifying requirements for other tasks.
  - The `callback` parameter is a `callback(err, result)` which must be called when finished, passing an `error` (which can be `null`) and the result of the function's execution. The remaining parameters name other tasks on which the task is dependent, and the results from those tasks are the arguments of those parameters.
- `callback(err, [results...])` - An optional callback which is called when all the tasks have been completed. It receives the `err` argument if any `tasks` pass an error to their callback. The remaining parameters are task names whose results you are interested in. This callback will only be called when all tasks have finished or an error has occurred, and so do not not specify dependencies in the same way as `tasks` do. If an error occurs, no further `tasks` will be performed, and `results` will only be valid for those tasks which managed to complete.

Example

The example from `auto` can be rewritten as follows:

```
async.autoInject({
    get_data: function(callback){
        // async code to get some data
        callback(null, 'data', 'converted to array');
    },
    make_folder: function(callback){
        // async code to create a directory to store a file in
        // this is run at the same time as getting the data
        callback(null, 'folder');
    },
    write_file: function(get_data, make_folder, callback){
        // once there is some data and the directory exists,
        // write the data to a file in the directory
        callback(null, 'filename');
    },
    email_link: function(write_file, callback){
        // once the file is written let's email a link to it...
        // write_file contains the filename returned by write_file.
        callback(null, {'file':write_file, 'email':'user@example.com'});
    }
}, function(err, email_link) {
    console.log('err = ', err);
    console.log('email_link = ', email_link);
});
```

If you are using a JS minifier that mangles parameter names, `autoInject` will not work with plain functions, since the parameter names will be collapsed to a single letter identifier. To work around this, you can explicitly specify the names of the parameters your task function needs in an array, similar to Angular.js dependency injection.

```
async.autoInject({
    //...
    write_file: ['get_data', 'make_folder', function(get_data, make_folder, callback){
        callback(null, 'filename');
    }],
    email_link: ['write_file', function(write_file, callback){
        callback(null, {'file':write_file, 'email':'user@example.com'});
    }]
    //...
}, done);
```

This still has an advantage over plain `auto`, since the results a task depends on are still spread into arguments.

---

retry([opts = {times: 5, interval: 0}| 5], task, [callback])

Attempts to get a successful response from `task` no more than `times` times before returning an error. If the task is successful, the `callback` will be passed the result of the successful task. If all attempts fail, the callback will be passed the error and result (if any) of the final attempt.

Arguments

- `opts` - Can be either an object with `times` and `interval` or a number.
  - `times` - The number of attempts to make before giving up. The default is `5`.
  - `interval` - The time to wait between retries, in milliseconds. The default is `0`.
  - If `opts` is a number, the number specifies the number of times to retry, with the default interval of `0`.
- `task(callback, results)` - A function which receives two arguments: (1) a `callback(err, result)` which must be called when finished, passing `err` (which can be `null`) and the `result` of the function's execution, and (2) a `results` object, containing the results of the previously executed functions (if nested inside another control flow).
- `callback(err, results)` - An optional callback which is called when the task has succeeded, or after the final failed attempt. It receives the `err` and `result` arguments of the last attempt at completing the `task`.

The `retry` function can be used as a stand-alone control flow by passing a callback, as shown below:

```
// try calling apiMethod 3 times
async.retry(3, apiMethod, function(err, result) {
    // do something with the result
});
```

```
// try calling apiMethod 3 times, waiting 200 ms between each retry
async.retry({times: 3, interval: 200}, apiMethod, function(err, result) {
    // do something with the result
});
```

```
// try calling apiMethod the default 5 times no delay between each retry
async.retry(apiMethod, function(err, result) {
    // do something with the result
});
```

It can also be embedded within other control flow functions to retry individual methods that are not as reliable, like this:

```
async.auto({
    users: api.getUsers.bind(api),
    payments: async.retry(3, api.getPayments.bind(api))
}, function(err, results) {
  // do something with the results
});
```

retryable([opts = {times: 5, interval: 0}| 5], task)

A close relative of `retry`. This method wraps a task and makes it retryable, rather than immediately calling it with retries.

Arguments

- `opts` - optional options, exactly the same as from `retry`
- `task` - the asynchronous function to wrap

Example

```
async.auto({
    dep1: async.retryable(3, getFromFlakyService),
    process: ["dep1", async.retryable(3, function (results, cb) {
        maybeProcessData(results.dep1, cb)
    })]
}, callback)
```

iterator(tasks)

Creates an iterator function which calls the next function in the `tasks` array, returning a continuation to call the next one after that. It's also possible to “peek” at the next iterator with `iterator.next()`.

This function is used internally by the `async` module, but can be useful when you want to manually control the flow of functions in series.

Arguments

- `tasks` - An array of functions to run.

Example

```
var iterator = async.iterator([
    function(){ sys.p('one'); },
    function(){ sys.p('two'); },
    function(){ sys.p('three'); }
]);

node> var iterator2 = iterator();
'one'
node> var iterator3 = iterator2();
'two'
node> iterator3();
'three'
node> var nextfn = iterator2.next();
node> nextfn();
'three'
```

Utils

apply(function, arguments..)

Creates a continuation function with some arguments already applied.

Useful as a shorthand when combined with other control flow functions. Any arguments passed to the returned function are added to the arguments originally passed to apply.

Arguments

- `function` - The function you want to eventually apply all arguments to.
- `arguments...` - Any number of arguments to automatically apply when the continuation is called.

Example

```
// using apply

async.parallel([
    async.apply(fs.writeFile, 'testfile1', 'test1'),
    async.apply(fs.writeFile, 'testfile2', 'test2'),
]);


// the same process without using apply

async.parallel([
    function(callback){
        fs.writeFile('testfile1', 'test1', callback);
    },
    function(callback){
        fs.writeFile('testfile2', 'test2', callback);
    }
]);
```

It's possible to pass any number of additional arguments when calling the
continuation:

```
node> var fn = async.apply(sys.puts, 'one');
node> fn('two', 'three');
one
two
three
```

nextTick(callback, [args...]), setImmediate(callback, [args...])

Calls `callback` on a later loop around the event loop. In Node.js this just calls `setImmediate`. In the browser it will use `setImmediate` if available, otherwise `setTimeout(callback, 0)`, which means other higher priority events may precede the execution of `callback`.

This is used internally for browser-compatibility purposes.

Arguments

- `callback` - The function to call on a later loop around the event loop.
- `args...` - any number of additional arguments to pass to the callback on the next tick

Example

```
var call_order = [];
async.nextTick(function(){
    call_order.push('two');
    // call_order now equals ['one','two']
});
call_order.push('one')

async.setImmediate(function (a, b, c) {
  // a, b, and c equal 1, 2, and 3
}, 1, 2, 3)
```

times(n, iteratee, [callback])

Calls the `iteratee` function `n` times, and accumulates results in the same manner
you would use with `map`.

Arguments

- `n` - The number of times to run the function.
- `iteratee` - The function to call `n` times.
- `callback` - see `map`

Example

```
// Pretend this is some complicated async factory
var createUser = function(id, callback) {
  callback(null, {
    id: 'user' + id
  })
}
// generate 5 users
async.times(5, function(n, next){
    createUser(n, function(err, user) {
      next(err, user)
    })
}, function(err, users) {
  // we should now have 5 users
});
```

Related

- timesSeries(n, iteratee, [callback])
- timesLimit(n, limit, iteratee, [callback])

race(tasks, [callback])

Runs the `tasks` array of functions in parallel, without waiting until the previous function has completed. Once any the `tasks` completed or pass an error to its callback, the main `callback` is immediately called. It's equivalent to `Promise.race()`.

Arguments

- `tasks` - An array containing functions to run. Each function is passed a `callback(err, result)` which it must call on completion with an error `err` (which can be `null`) and an optional `result` value.
- `callback(err, result)` - A callback to run once any of the functions have completed. This function gets an error or result from the first function that completed.

Example

```
async.race([
    function(callback){
        setTimeout(function(){
            callback(null, 'one');
        }, 200);
    },
    function(callback){
        setTimeout(function(){
            callback(null, 'two');
        }, 100);
    }
],
// main callback
function(err, result){
    // the result will be equal to 'two' as it finishes earlier
});
```

memoize(fn, [hasher])

Caches the results of an `async` function. When creating a hash to store function results against, the callback is omitted from the hash and an optional hash function can be used.

If no hash function is specified, the first argument is used as a hash key, which may work reasonably if it is a string or a data type that converts to a distinct string. Note that objects and arrays will not behave reasonably. Neither will cases where the other arguments are significant. In such cases, specify your own hash function.

The cache of results is exposed as the `memo` property of the function returned by `memoize`.

Arguments

- `fn` - The function to proxy and cache results from.
- `hasher` - An optional function for generating a custom hash for storing results. It has all the arguments applied to it apart from the callback, and must be synchronous.

Example

```
var slow_fn = function (name, callback) {
    // do something
    callback(null, result);
};
var fn = async.memoize(slow_fn);

// fn can now be used as if it were slow_fn
fn('some name', function () {
    // callback
});
```

unmemoize(fn)

Undoes a `memoize`d function, reverting it to the original, unmemoized form. Handy for testing.

Arguments

- `fn` - the memoized function

ensureAsync(fn)

Wrap an async function and ensure it calls its callback on a later tick of the event loop. If the function already calls its callback on a next tick, no extra deferral is added. This is useful for preventing stack overflows (`RangeError: Maximum call stack size exceeded`) and generally keeping Zalgo contained.

Arguments

- `fn` - an async function, one that expects a node-style callback as its last argument

Returns a wrapped function with the exact same call signature as the function passed in.

Example

```
function sometimesAsync(arg, callback) {
    if (cache[arg]) {
        return callback(null, cache[arg]); // this would be synchronous!!
    } else {
       doSomeIO(arg, callback); // this IO would be asynchronous
    }
}

// this has a risk of stack overflows if many results are cached in a row
async.mapSeries(args, sometimesAsync, done);

// this will defer sometimesAsync's callback if necessary,
// preventing stack overflows
async.mapSeries(args, async.ensureAsync(sometimesAsync), done);
```

constant(values...)

Returns a function that when called, calls-back with the values provided. Useful as the first function in a `waterfall`, or for plugging values in to `auto`.

Example

```
async.waterfall([
    async.constant(42),
    function (value, next) {
        // value === 42
    },
    //...
], callback);

async.waterfall([
    async.constant(filename, "utf8"),
    fs.readFile,
    function (fileData, next) {
        //...
    }
    //...
], callback);

async.auto({
    hostname: async.constant("https://server.net/"),
    port: findFreePort,
    launchServer: ["hostname", "port", function (options, cb) {
        startServer(options, cb);
    }],
    //...
}, callback);
```

asyncify(func)

Alias: `wrapSync`

Take a sync function and make it async, passing its return value to a callback. This is useful for plugging sync functions into a waterfall, series, or other async functions. Any arguments passed to the generated function will be passed to the wrapped function (except for the final callback argument). Errors thrown will be passed to the callback.

Example

```
async.waterfall([
    async.apply(fs.readFile, filename, "utf8"),
    async.asyncify(JSON.parse),
    function (data, next) {
        // data is the result of parsing the text.
        // If there was a parsing error, it would have been caught.
    }
], callback)
```

If the function passed to `asyncify` returns a Promise, that promises's resolved/rejected state will be used to call the callback, rather than simply the synchronous return value. Example:

```
async.waterfall([
    async.apply(fs.readFile, filename, "utf8"),
    async.asyncify(function (contents) {
        return db.model.create(contents);
    }),
    function (model, next) {
        // `model` is the instantiated model object.
        // If there was an error, this function would be skipped.
    }
], callback)
```

This also means you can asyncify ES2016 `async` functions.

```
var q = async.queue(async.asyncify(async function (file) {
    var intermediateStep = await processFile(file);
    return await somePromise(intermediateStep)
}));

q.push(files);
```

log(function, arguments)

Logs the result of an `async` function to the `console`. Only works in Node.js or
in browsers that support `console.log` and `console.error` (such as FF and Chrome).
If multiple arguments are returned from the async function, `console.log` is
called on each argument in order.

Arguments

- `function` - The function you want to eventually apply all arguments to.
- `arguments...` - Any number of arguments to apply to the function.

Example

```
var hello = function(name, callback){
    setTimeout(function(){
        callback(null, 'hello ' + name);
    }, 1000);
};
```

```
node> async.log(hello, 'world');
'hello world'
```

dir(function, arguments)

Logs the result of an `async` function to the `console` using `console.dir` to
display the properties of the resulting object. Only works in Node.js or
in browsers that support `console.dir` and `console.error` (such as FF and Chrome).
If multiple arguments are returned from the async function, `console.dir` is
called on each argument in order.

Arguments

- `function` - The function you want to eventually apply all arguments to.
- `arguments...` - Any number of arguments to apply to the function.

Example

```
var hello = function(name, callback){
    setTimeout(function(){
        callback(null, {hello: name});
    }, 1000);
};
```

```
node> async.dir(hello, 'world');
{hello: 'world'}
```

noConflict()

Changes the value of `async` back to its original value, returning a reference to the
`async` object.

timeout(function, miliseconds)

Sets a time limit on an asynchronous function. If the function does not call its callback within the specified miliseconds, it will be called with a timeout error. The code property for the error object will be `'ETIMEDOUT'`.

Returns a wrapped function that can be used with any of the control flow functions.

Arguments

- `function` - The asynchronous function you want to set the time limit.
- `miliseconds` - The specified time limit.

Example

```
async.timeout(function(callback) {
  doAsyncTask(callback);
}, 1000);
```