Domain

Stability: 0 - Deprecated

This module is pending deprecation. Once a replacement API has been finalized, this module will be fully deprecated. Most developers should **not** have cause to use this module. Users who absolutely must have the functionality that domains provide may rely on it for the time being but should expect to have to migrate to a different solution in the future.

Domains provide a way to handle multiple different IO operations as a single group. If any of the event emitters or callbacks registered to a domain emit an `'error'` event, or throw an error, then the domain object will be notified, rather than losing the context of the error in the `process.on('uncaughtException')` handler, or causing the program to exit immediately with an error code.

Warning: Don't ignore errors!

Domain error handlers are not a substitute for closing down a process when an error occurs.

By the very nature of how `throw` works in JavaScript, there is almost never any way to safely "pick up where it left off", without leaking references, or creating some other sort of undefined brittle state.

The safest way to respond to a thrown error is to shut down the process. Of course, in a normal web server, there may be many open connections, and it is not reasonable to abruptly shut those down because an error was triggered by someone else.

The better approach is to send an error response to the request that triggered the error, while letting the others finish in their normal time, and stop listening for new requests in that worker.

In this way, `domain` usage goes hand-in-hand with the cluster module, since the primary process can fork a new worker when a worker encounters an error. For Node.js programs that scale to multiple machines, the terminating proxy or service registry can take note of the failure, and react accordingly.

For example, this is not a good idea:

```
// XXX WARNING! BAD IDEA!

const d = require('domain').create();
d.on('error', (er) => {
  // The error won't crash the process, but what it does is worse!
  // Though we've prevented abrupt process restarting, we are leaking
  // a lot of resources if this ever happens.
  // This is no better than process.on('uncaughtException')!
  console.log(`error, but oh well ${er.message}`);
```

```
});
d.run(() => {
  require('http').createServer((req, res) => {
    handleRequest(req, res);
  }).listen(PORT);
});
```

By using the context of a domain, and the resilience of separating our program into multiple worker processes, we can react more appropriately, and handle errors with much greater safety.

```
// Much better!

const cluster = require('cluster');
const PORT = +process.env.PORT || 1337;

if (cluster.isPrimary) {
  // A more realistic scenario would have more than 2 workers,
  // and perhaps not put the primary and worker in the same file.
  //
  // It is also possible to get a bit fancier about logging, and
  // implement whatever custom logic is needed to prevent DoS
  // attacks and other bad behavior.
  //
  // See the options in the cluster documentation.
  //
  // The important thing is that the primary does very little,
  // increasing our resilience to unexpected errors.

  cluster.fork();
  cluster.fork();

  cluster.on('disconnect', (worker) => {
    console.error('disconnect!');
    cluster.fork();
  });

} else {
  // the worker
  //
  // This is where we put our bugs!

  const domain = require('domain');

  // See the cluster documentation for more details about using
  // worker processes to serve requests. How it works, caveats, etc.
```

```
const server = require('http').createServer((req, res) => {
  const d = domain.create();
  d.on('error', (er) => {
    console.error(`error ${er.stack}`);

    // We're in dangerous territory!
    // By definition, something unexpected occurred,
    // which we probably didn't want.
    // Anything can happen now! Be very careful!

    try {
      // Make sure we close down within 30 seconds
      const killtimer = setTimeout(() => {
        process.exit(1);
      }, 30000);
      // But don't keep the process open just for that!
      killtimer.unref();

      // Stop taking new requests.
      server.close();

      // Let the primary know we're dead. This will trigger a
      // 'disconnect' in the cluster primary, and then it will fork
      // a new worker.
      cluster.worker.disconnect();

      // Try to send an error to the request that triggered the problem
      res.statusCode = 500;
      res.setHeader('content-type', 'text/plain');
      res.end('Oops, there was a problem!\n');
    } catch (er2) {
      // Oh well, not much we can do at this point.
      console.error(`Error sending 500! ${er2.stack}`);
    }
  });

  // Because req and res were created before this domain existed,
  // we need to explicitly add them.
  // See the explanation of implicit vs explicit binding below.
  d.add(req);
  d.add(res);

  // Now run the handler function in the domain.
  d.run(() => {
    handleRequest(req, res);
  });
```

```
  });
  server.listen(PORT);
}

// This part is not important. Just an example routing thing.
// Put fancy application logic here.
function handleRequest(req, res) {
  switch (req.url) {
    case '/error':
      // We do some async stuff, and then...
      setTimeout(() => {
        // Whoops!
        flerb.bark();
      }, timeout);
      break;
    default:
      res.end('ok');
  }
}
```

Additions to `Error` objects

Any time an `Error` object is routed through a domain, a few extra fields are added to it.

- `error.domain` The domain that first handled the error.
- `error.domainEmitter` The event emitter that emitted an `'error'` event with the error object.
- `error.domainBound` The callback function which was bound to the domain, and passed an error as its first argument.
- `error.domainThrown` A boolean indicating whether the error was thrown, emitted, or passed to a bound callback function.

Implicit binding

If domains are in use, then all **new** `EventEmitter` objects (including Stream objects, requests, responses, etc.) will be implicitly bound to the active domain at the time of their creation.

Additionally, callbacks passed to lowlevel event loop requests (such as to `fs.open()`, or other callback-taking methods) will automatically be bound to the active domain. If they throw, then the domain will catch the error.

In order to prevent excessive memory usage, `Domain` objects themselves are not implicitly added as children of the active domain. If they were, then it would be too easy to prevent request and response objects from being properly garbage collected.

To nest `Domain` objects as children of a parent `Domain` they must be explicitly added.

Implicit binding routes thrown errors and `'error'` events to the `Domain`'s `'error'` event, but does not register the `EventEmitter` on the `Domain`. Implicit binding only takes care of thrown errors and `'error'` events.

Explicit binding

Sometimes, the domain in use is not the one that ought to be used for a specific event emitter. Or, the event emitter could have been created in the context of one domain, but ought to instead be bound to some other domain.

For example, there could be one domain in use for an HTTP server, but perhaps we would like to have a separate domain to use for each request.

That is possible via explicit binding.

```
// Create a top-level domain for the server
const domain = require('domain');
const http = require('http');
const serverDomain = domain.create();

serverDomain.run(() => {
  // Server is created in the scope of serverDomain
  http.createServer((req, res) => {
    // Req and res are also created in the scope of serverDomain
    // however, we'd prefer to have a separate domain for each request.
    // create it first thing, and add req and res to it.
    const reqd = domain.create();
    reqd.add(req);
    reqd.add(res);
    reqd.on('error', (er) => {
      console.error('Error', er, req.url);
      try {
        res.writeHead(500);
        res.end('Error occurred, sorry.');
      } catch (er2) {
        console.error('Error sending 500', er2, req.url);
      }
    });
  }).listen(1337);
});
```

`domain.create()`

- Returns: {Domain}

Class: `Domain`

- Extends: {EventEmitter}

The `Domain` class encapsulates the functionality of routing errors and uncaught exceptions to the active `Domain` object.

To handle the errors that it catches, listen to its `'error'` event.

`domain.members`

- {Array}

An array of timers and event emitters that have been explicitly added to the domain.

`domain.add(emitter)`

- `emitter` {EventEmitter|Timer} emitter or timer to be added to the domain

Explicitly adds an emitter to the domain. If any event handlers called by the emitter throw an error, or if the emitter emits an `'error'` event, it will be routed to the domain's `'error'` event, just like with implicit binding.

This also works with timers that are returned from `setInterval()` and `setTimeout()`. If their callback function throws, it will be caught by the domain `'error'` handler.

If the Timer or `EventEmitter` was already bound to a domain, it is removed from that one, and bound to this one instead.

`domain.bind(callback)`

- `callback` {Function} The callback function
- Returns: {Function} The bound function

The returned function will be a wrapper around the supplied callback function. When the returned function is called, any errors that are thrown will be routed to the domain's `'error'` event.

```
const d = domain.create();

function readSomeFile(filename, cb) {
  fs.readFile(filename, 'utf8', d.bind((er, data) => {
    // If this throws, it will also be passed to the domain.
    return cb(er, data ? JSON.parse(data) : null);
  }));
}

d.on('error', (er) => {
  // An error occurred somewhere. If we throw it now, it will crash the program
```

```
  // with the normal line number and stack message.
});
```

`domain.enter()`

The `enter()` method is plumbing used by the `run()`, `bind()`, and `intercept()` methods to set the active domain. It sets `domain.active` and `process.domain` to the domain, and implicitly pushes the domain onto the domain stack managed by the domain module (see `domain.exit()` for details on the domain stack). The call to `enter()` delimits the beginning of a chain of asynchronous calls and I/O operations bound to a domain.

Calling `enter()` changes only the active domain, and does not alter the domain itself. `enter()` and `exit()` can be called an arbitrary number of times on a single domain.

`domain.exit()`

The `exit()` method exits the current domain, popping it off the domain stack. Any time execution is going to switch to the context of a different chain of asynchronous calls, it's important to ensure that the current domain is exited. The call to `exit()` delimits either the end of or an interruption to the chain of asynchronous calls and I/O operations bound to a domain.

If there are multiple, nested domains bound to the current execution context, `exit()` will exit any domains nested within this domain.

Calling `exit()` changes only the active domain, and does not alter the domain itself. `enter()` and `exit()` can be called an arbitrary number of times on a single domain.

`domain.intercept(callback)`

- `callback` {Function} The callback function
- Returns: {Function} The intercepted function

This method is almost identical to `domain.bind(callback)`. However, in addition to catching thrown errors, it will also intercept `Error` objects sent as the first argument to the function.

In this way, the common `if (err) return callback(err);` pattern can be replaced with a single error handler in a single place.

```
const d = domain.create();

function readSomeFile(filename, cb) {
  fs.readFile(filename, 'utf8', d.intercept((data) => {
    // Note, the first argument is never passed to the
    // callback since it is assumed to be the 'Error' argument
```

```
    // and thus intercepted by the domain.

    // If this throws, it will also be passed to the domain
    // so the error-handling logic can be moved to the 'error'
    // event on the domain instead of being repeated throughout
    // the program.
    return cb(null, JSON.parse(data));
  }));
}

d.on('error', (er) => {
  // An error occurred somewhere. If we throw it now, it will crash the program
  // with the normal line number and stack message.
});
```

`domain.remove(emitter)`

- `emitter` {EventEmitter|Timer} emitter or timer to be removed from the domain

The opposite of `domain.add(emitter)`. Removes domain handling from the specified emitter.

`domain.run(fn[, ...args])`

- `fn` {Function}
- `...args` {any}

Run the supplied function in the context of the domain, implicitly binding all event emitters, timers, and lowlevel requests that are created in that context. Optionally, arguments can be passed to the function.

This is the most basic way to use a domain.

```
const domain = require('domain');
const fs = require('fs');
const d = domain.create();
d.on('error', (er) => {
  console.error('Caught error!', er);
});
d.run(() => {
  process.nextTick(() => {
    setTimeout(() => { // Simulating some various async stuff
      fs.open('non-existent file', 'r', (er, fd) => {
        if (er) throw er;
        // proceed...
      });
    }, 100);
```

```
  });
});
```

In this example, the `d.on('error')` handler will be triggered, rather than crashing the program.

Domains and promises

As of Node.js 8.0.0, the handlers of promises are run inside the domain in which the call to `.then()` or `.catch()` itself was made:

```
const d1 = domain.create();
const d2 = domain.create();

let p;
d1.run(() => {
  p = Promise.resolve(42);
});

d2.run(() => {
  p.then((v) => {
    // running in d2
  });
});
```

A callback may be bound to a specific domain using `domain.bind(callback)`:

```
const d1 = domain.create();
const d2 = domain.create();

let p;
d1.run(() => {
  p = Promise.resolve(42);
});

d2.run(() => {
  p.then(p.domain.bind((v) => {
    // running in d1
  }));
});
```

Domains will not interfere with the error handling mechanisms for promises. In other words, no `'error'` event will be emitted for unhandled `Promise` rejections.