Asynchronous context tracking

> *Stability: 2 - Stable*

Introduction

These classes are used to associate state and propagate it throughout callbacks and promise chains. They allow storing data throughout the lifetime of a web request or any other asynchronous duration. It is similar to thread-local storage in other languages.

The `AsyncLocalStorage` and `AsyncResource` classes are part of the `async_hooks` module:

```
import { AsyncLocalStorage, AsyncResource } from 'async_hooks';
```

```
const { AsyncLocalStorage, AsyncResource } = require('async_hooks');
```

Class: `AsyncLocalStorage`

This class creates stores that stay coherent through asynchronous operations.

While you can create your own implementation on top of the `async_hooks` module, `AsyncLocalStorage` should be preferred as it is a performant and memory safe implementation that involves significant optimizations that are non-obvious to implement.

The following example uses `AsyncLocalStorage` to build a simple logger that assigns IDs to incoming HTTP requests and includes them in messages logged within each request.

```
import http from 'http';
import { AsyncLocalStorage } from 'async_hooks';

const asyncLocalStorage = new AsyncLocalStorage();

function logWithId(msg) {
  const id = asyncLocalStorage.getStore();
  console.log(`${id !== undefined ? id : '-'}:`, msg);
}

let idSeq = 0;
http.createServer((req, res) => {
  asyncLocalStorage.run(idSeq++, () => {
    logWithId('start');
    // Imagine any chain of async operations here
    setImmediate(() => {
      logWithId('finish');
      res.end();
    });
  });
}).listen(8080);
```

```
http.get('http://localhost:8080');
http.get('http://localhost:8080');
// Prints:
//   0: start
//   1: start
//   0: finish
//   1: finish
```

```
const http = require('http');
const { AsyncLocalStorage } = require('async_hooks');

const asyncLocalStorage = new AsyncLocalStorage();

function logWithId(msg) {
  const id = asyncLocalStorage.getStore();
  console.log(`${id !== undefined ? id : '-'}:`, msg);
}

let idSeq = 0;
http.createServer((req, res) => {
  asyncLocalStorage.run(idSeq++, () => {
    logWithId('start');
    // Imagine any chain of async operations here
    setImmediate(() => {
      logWithId('finish');
      res.end();
    });
  });
}).listen(8080);

http.get('http://localhost:8080');
http.get('http://localhost:8080');
// Prints:
//   0: start
//   1: start
//   0: finish
//   1: finish
```

Each instance of `AsyncLocalStorage` maintains an independent storage context. Multiple instances can safely exist simultaneously without risk of interfering with each other's data.

`new AsyncLocalStorage()`

Creates a new instance of `AsyncLocalStorage`. Store is only provided within a `run()` call or after an `enterWith()` call.

`asyncLocalStorage.disable()`

> *Stability: 1 - Experimental*

Disables the instance of `AsyncLocalStorage`. All subsequent calls to `asyncLocalStorage.getStore()` will return `undefined` until `asyncLocalStorage.run()` or `asyncLocalStorage.enterWith()` is called again.

When calling `asyncLocalStorage.disable()`, all current contexts linked to the instance will be exited.

Calling `asyncLocalStorage.disable()` is required before the `asyncLocalStorage` can be garbage collected. This does not apply to stores provided by the `asyncLocalStorage`, as those objects are garbage collected along with the corresponding async resources.

Use this method when the `asyncLocalStorage` is not in use anymore in the current process.

`asyncLocalStorage.getStore()`

- Returns: {any}

Returns the current store. If called outside of an asynchronous context initialized by calling `asyncLocalStorage.run()` or `asyncLocalStorage.enterWith()`, it returns `undefined`.

`asyncLocalStorage.enterWith(store)`

> *Stability: 1 - Experimental*

- `store` {any}

Transitions into the context for the remainder of the current synchronous execution and then persists the store through any following asynchronous calls.

Example:

```
const store = { id: 1 };
// Replaces previous store with the given store object
asyncLocalStorage.enterWith(store);
asyncLocalStorage.getStore(); // Returns the store object
someAsyncOperation(() => {
  asyncLocalStorage.getStore(); // Returns the same object
});
```

This transition will continue for the *entire* synchronous execution. This means that if, for example, the context is entered within an event handler subsequent event handlers will also run within that context unless specifically bound to another context with an `AsyncResource`. That is why `run()` should be preferred over `enterWith()` unless there are strong reasons to use the latter method.

```
const store = { id: 1 };

emitter.on('my-event', () => {
  asyncLocalStorage.enterWith(store);
});
emitter.on('my-event', () => {
  asyncLocalStorage.getStore(); // Returns the same object
});

asyncLocalStorage.getStore(); // Returns undefined
```

```
emitter.emit('my-event');
asyncLocalStorage.getStore(); // Returns the same object
```

`asyncLocalStorage.run(store, callback[, ...args])`

- `store` {any}
- `callback` {Function}
- `...args` {any}

Runs a function synchronously within a context and returns its return value. The store is not accessible outside of the callback function. The store is accessible to any asynchronous operations created within the callback.

The optional `args` are passed to the callback function.

If the callback function throws an error, the error is thrown by `run()` too. The stacktrace is not impacted by this call and the context is exited.

Example:

```
const store = { id: 2 };
try {
  asyncLocalStorage.run(store, () => {
    asyncLocalStorage.getStore(); // Returns the store object
    setTimeout(() => {
      asyncLocalStorage.getStore(); // Returns the store object
    }, 200);
    throw new Error();
  });
} catch (e) {
  asyncLocalStorage.getStore(); // Returns undefined
  // The error will be caught here
}
```

`asyncLocalStorage.exit(callback[, ...args])`

> *Stability: 1 - Experimental*

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

Runs a function synchronously outside of a context and returns its return value. The store is not accessible within the callback function or the asynchronous operations created within the callback. Any `getStore()` call done within the callback function will always return `undefined`.

The optional `args` are passed to the callback function.

If the callback function throws an error, the error is thrown by `exit()` too. The stacktrace is not impacted by this call and the context is re-entered.

Example:

```
// Within a call to run
try {
```

```
  asyncLocalStorage.getStore(); // Returns the store object or value
  asyncLocalStorage.exit(() => {
    asyncLocalStorage.getStore(); // Returns undefined
    throw new Error();
  });
} catch (e) {
  asyncLocalStorage.getStore(); // Returns the same object or value
  // The error will be caught here
}
```

Usage with `async/await`

If, within an async function, only one `await` call is to run within a context, the following pattern should be used:

```
async function fn() {
  await asyncLocalStorage.run(new Map(), () => {
    asyncLocalStorage.getStore().set('key', value);
    return foo(); // The return value of foo will be awaited
  });
}
```

In this example, the store is only available in the callback function and the functions called by `foo`. Outside of `run`, calling `getStore` will return `undefined`.

Troubleshooting: Context loss

In most cases, `AsyncLocalStorage` works without issues. In rare situations, the current store is lost in one of the asynchronous operations.

If your code is callback-based, it is enough to promisify it with `util.promisify()` so it starts working with native promises.

If you need to use a callback-based API or your code assumes a custom thenable implementation, use the `AsyncResource` class to associate the asynchronous operation with the correct execution context. Find the function call responsible for the context loss by logging the content of `asyncLocalStorage.getStore()` after the calls you suspect are responsible for the loss. When the code logs `undefined`, the last callback called is probably responsible for the context loss.

Class: `AsyncResource`

The class `AsyncResource` is designed to be extended by the embedder's async resources. Using this, users can easily trigger the lifetime events of their own resources.

The `init` hook will trigger when an `AsyncResource` is instantiated.

The following is an overview of the `AsyncResource` API.

```
import { AsyncResource, executionAsyncId } from 'async_hooks';

// AsyncResource() is meant to be extended. Instantiating a
// new AsyncResource() also triggers init. If triggerAsyncId is omitted then
```

```
// async_hook.executionAsyncId() is used.
const asyncResource = new AsyncResource(
  type, { triggerAsyncId: executionAsyncId(), requireManualDestroy: false }
);

// Run a function in the execution context of the resource. This will
// * establish the context of the resource
// * trigger the AsyncHooks before callbacks
// * call the provided function `fn` with the supplied arguments
// * trigger the AsyncHooks after callbacks
// * restore the original execution context
asyncResource.runInAsyncScope(fn, thisArg, ...args);

// Call AsyncHooks destroy callbacks.
asyncResource.emitDestroy();

// Return the unique ID assigned to the AsyncResource instance.
asyncResource.asyncId();

// Return the trigger ID for the AsyncResource instance.
asyncResource.triggerAsyncId();
```

```
const { AsyncResource, executionAsyncId } = require('async_hooks');

// AsyncResource() is meant to be extended. Instantiating a
// new AsyncResource() also triggers init. If triggerAsyncId is omitted then
// async_hook.executionAsyncId() is used.
const asyncResource = new AsyncResource(
  type, { triggerAsyncId: executionAsyncId(), requireManualDestroy: false }
);

// Run a function in the execution context of the resource. This will
// * establish the context of the resource
// * trigger the AsyncHooks before callbacks
// * call the provided function `fn` with the supplied arguments
// * trigger the AsyncHooks after callbacks
// * restore the original execution context
asyncResource.runInAsyncScope(fn, thisArg, ...args);

// Call AsyncHooks destroy callbacks.
asyncResource.emitDestroy();

// Return the unique ID assigned to the AsyncResource instance.
asyncResource.asyncId();

// Return the trigger ID for the AsyncResource instance.
asyncResource.triggerAsyncId();
```

`new AsyncResource(type[, options])`

- `type` {string} The type of async event.

- `options` {Object}
  - `triggerAsyncId` {number} The ID of the execution context that created this async event. **Default:** `executionAsyncId()`.
  - `requireManualDestroy` {boolean} If set to `true`, disables `emitDestroy` when the object is garbage collected. This usually does not need to be set (even if `emitDestroy` is called manually), unless the resource's `asyncId` is retrieved and the sensitive API's `emitDestroy` is called with it. When set to `false`, the `emitDestroy` call on garbage collection will only take place if there is at least one active `destroy` hook. **Default:** `false`.

Example usage:

```
class DBQuery extends AsyncResource {
  constructor(db) {
    super('DBQuery');
    this.db = db;
  }

  getInfo(query, callback) {
    this.db.get(query, (err, data) => {
      this.runInAsyncScope(callback, null, err, data);
    });
  }

  close() {
    this.db = null;
    this.emitDestroy();
  }
}
```

Static method: `AsyncResource.bind(fn[, type, [thisArg]])`

- `fn` {Function} The function to bind to the current execution context.
- `type` {string} An optional name to associate with the underlying `AsyncResource`.
- `thisArg` {any}

Binds the given function to the current execution context.

The returned function will have an `asyncResource` property referencing the `AsyncResource` to which the function is bound.

`asyncResource.bind(fn[, thisArg])`

- `fn` {Function} The function to bind to the current `AsyncResource`.
- `thisArg` {any}

Binds the given function to execute to this `AsyncResource`'s scope.

The returned function will have an `asyncResource` property referencing the `AsyncResource` to which the function is bound.

`asyncResource.runInAsyncScope(fn[, thisArg, ...args])`

- `fn` {Function} The function to call in the execution context of this async resource.
- `thisArg` {any} The receiver to be used for the function call.
- `...args` {any} Optional arguments to pass to the function.

Call the provided function with the provided arguments in the execution context of the async resource. This will establish the context, trigger the AsyncHooks before callbacks, call the function, trigger the AsyncHooks after callbacks, and then restore the original execution context.

`asyncResource.emitDestroy()`

- Returns: {AsyncResource} A reference to `asyncResource`.

Call all `destroy` hooks. This should only ever be called once. An error will be thrown if it is called more than once. This **must** be manually called. If the resource is left to be collected by the GC then the `destroy` hooks will never be called.

`asyncResource.asyncId()`

- Returns: {number} The unique `asyncId` assigned to the resource.

`asyncResource.triggerAsyncId()`

- Returns: {number} The same `triggerAsyncId` that is passed to the `AsyncResource` constructor.

Using `AsyncResource` for a `Worker` thread pool

The following example shows how to use the `AsyncResource` class to properly provide async tracking for a `Worker` pool. Other resource pools, such as database connection pools, can follow a similar model.

Assuming that the task is adding two numbers, using a file named `task_processor.js` with the following content:

```mjs
import { parentPort } from 'worker_threads';
parentPort.on('message', (task) => {
  parentPort.postMessage(task.a + task.b);
});
```

```cjs
const { parentPort } = require('worker_threads');
parentPort.on('message', (task) => {
  parentPort.postMessage(task.a + task.b);
});
```

a Worker pool around it could use the following structure:

```mjs
import { AsyncResource } from 'async_hooks';
import { EventEmitter } from 'events';
import path from 'path';
import { Worker } from 'worker_threads';

const kTaskInfo = Symbol('kTaskInfo');
const kWorkerFreedEvent = Symbol('kWorkerFreedEvent');
```

```
class WorkerPoolTaskInfo extends AsyncResource {
  constructor(callback) {
    super('WorkerPoolTaskInfo');
    this.callback = callback;
  }

  done(err, result) {
    this.runInAsyncScope(this.callback, null, err, result);
    this.emitDestroy();  // `TaskInfo`s are used only once.
  }
}

export default class WorkerPool extends EventEmitter {
  constructor(numThreads) {
    super();
    this.numThreads = numThreads;
    this.workers = [];
    this.freeWorkers = [];
    this.tasks = [];

    for (let i = 0; i < numThreads; i++)
      this.addNewWorker();

    // Any time the kWorkerFreedEvent is emitted, dispatch
    // the next task pending in the queue, if any.
    this.on(kWorkerFreedEvent, () => {
      if (this.tasks.length > 0) {
        const { task, callback } = this.tasks.shift();
        this.runTask(task, callback);
      }
    });
  }

  addNewWorker() {
    const worker = new Worker(new URL('task_processer.js', import.meta.url));
    worker.on('message', (result) => {
      // In case of success: Call the callback that was passed to `runTask`,
      // remove the `TaskInfo` associated with the Worker, and mark it as free
      // again.
      worker[kTaskInfo].done(null, result);
      worker[kTaskInfo] = null;
      this.freeWorkers.push(worker);
      this.emit(kWorkerFreedEvent);
    });
    worker.on('error', (err) => {
      // In case of an uncaught exception: Call the callback that was passed to
      // `runTask` with the error.
      if (worker[kTaskInfo])
        worker[kTaskInfo].done(err, null);
      else
        this.emit('error', err);
      // Remove the worker from the list and start a new Worker to replace the
```

```
      // current one.
      this.workers.splice(this.workers.indexOf(worker), 1);
      this.addNewWorker();
    });
    this.workers.push(worker);
    this.freeWorkers.push(worker);
    this.emit(kWorkerFreedEvent);
  }

  runTask(task, callback) {
    if (this.freeWorkers.length === 0) {
      // No free threads, wait until a worker thread becomes free.
      this.tasks.push({ task, callback });
      return;
    }

    const worker = this.freeWorkers.pop();
    worker[kTaskInfo] = new WorkerPoolTaskInfo(callback);
    worker.postMessage(task);
  }

  close() {
    for (const worker of this.workers) worker.terminate();
  }
}
```

```
const { AsyncResource } = require('async_hooks');
const { EventEmitter } = require('events');
const path = require('path');
const { Worker } = require('worker_threads');

const kTaskInfo = Symbol('kTaskInfo');
const kWorkerFreedEvent = Symbol('kWorkerFreedEvent');

class WorkerPoolTaskInfo extends AsyncResource {
  constructor(callback) {
    super('WorkerPoolTaskInfo');
    this.callback = callback;
  }

  done(err, result) {
    this.runInAsyncScope(this.callback, null, err, result);
    this.emitDestroy();  // `TaskInfo`s are used only once.
  }
}

class WorkerPool extends EventEmitter {
  constructor(numThreads) {
    super();
    this.numThreads = numThreads;
    this.workers = [];
```

```
  this.freeWorkers = [];
  this.tasks = [];

  for (let i = 0; i < numThreads; i++)
    this.addNewWorker();

  // Any time the kWorkerFreedEvent is emitted, dispatch
  // the next task pending in the queue, if any.
  this.on(kWorkerFreedEvent, () => {
    if (this.tasks.length > 0) {
      const { task, callback } = this.tasks.shift();
      this.runTask(task, callback);
    }
  });
}

addNewWorker() {
  const worker = new Worker(path.resolve(__dirname, 'task_processor.js'));
  worker.on('message', (result) => {
    // In case of success: Call the callback that was passed to `runTask`,
    // remove the `TaskInfo` associated with the Worker, and mark it as free
    // again.
    worker[kTaskInfo].done(null, result);
    worker[kTaskInfo] = null;
    this.freeWorkers.push(worker);
    this.emit(kWorkerFreedEvent);
  });
  worker.on('error', (err) => {
    // In case of an uncaught exception: Call the callback that was passed to
    // `runTask` with the error.
    if (worker[kTaskInfo])
      worker[kTaskInfo].done(err, null);
    else
      this.emit('error', err);
    // Remove the worker from the list and start a new Worker to replace the
    // current one.
    this.workers.splice(this.workers.indexOf(worker), 1);
    this.addNewWorker();
  });
  this.workers.push(worker);
  this.freeWorkers.push(worker);
  this.emit(kWorkerFreedEvent);
}

runTask(task, callback) {
  if (this.freeWorkers.length === 0) {
    // No free threads, wait until a worker thread becomes free.
    this.tasks.push({ task, callback });
    return;
  }

  const worker = this.freeWorkers.pop();
```

```
    worker[kTaskInfo] = new WorkerPoolTaskInfo(callback);
    worker.postMessage(task);
  }

  close() {
    for (const worker of this.workers) worker.terminate();
  }
}

module.exports = WorkerPool;
```

Without the explicit tracking added by the `WorkerPoolTaskInfo` objects, it would appear that the callbacks are associated with the individual `Worker` objects. However, the creation of the `Worker`s is not associated with the creation of the tasks and does not provide information about when tasks were scheduled.

This pool could be used as follows:

```
import WorkerPool from './worker_pool.js';
import os from 'os';

const pool = new WorkerPool(os.cpus().length);

let finished = 0;
for (let i = 0; i < 10; i++) {
  pool.runTask({ a: 42, b: 100 }, (err, result) => {
    console.log(i, err, result);
    if (++finished === 10)
      pool.close();
  });
}
```

```
const WorkerPool = require('./worker_pool.js');
const os = require('os');

const pool = new WorkerPool(os.cpus().length);

let finished = 0;
for (let i = 0; i < 10; i++) {
  pool.runTask({ a: 42, b: 100 }, (err, result) => {
    console.log(i, err, result);
    if (++finished === 10)
      pool.close();
  });
}
```

Integrating `AsyncResource` with `EventEmitter`

Event listeners triggered by an `EventEmitter` may be run in a different execution context than the one that was active when `eventEmitter.on()` was called.

The following example shows how to use the `AsyncResource` class to properly associate an event listener with the correct execution context. The same approach can be applied to a `Stream` or a similar event-driven class.

```mjs
import { createServer } from 'http';
import { AsyncResource, executionAsyncId } from 'async_hooks';

const server = createServer((req, res) => {
  req.on('close', AsyncResource.bind(() => {
    // Execution context is bound to the current outer scope.
  }));
  req.on('close', () => {
    // Execution context is bound to the scope that caused 'close' to emit.
  });
  res.end();
}).listen(3000);
```

```cjs
const { createServer } = require('http');
const { AsyncResource, executionAsyncId } = require('async_hooks');

const server = createServer((req, res) => {
  req.on('close', AsyncResource.bind(() => {
    // Execution context is bound to the current outer scope.
  }));
  req.on('close', () => {
    // Execution context is bound to the scope that caused 'close' to emit.
  });
  res.end();
}).listen(3000);
```