Async hooks

> *Stability: 1 - Experimental*

The `async_hooks` module provides an API to track asynchronous resources. It can be accessed using:

```
import async_hooks from 'async_hooks';
```

```
const async_hooks = require('async_hooks');
```

Terminology

An asynchronous resource represents an object with an associated callback. This callback may be called multiple times, such as the `'connection'` event in `net.createServer()`, or just a single time like in `fs.open()`. A resource can also be closed before the callback is called. `AsyncHook` does not explicitly distinguish between these different cases but will represent them as the abstract concept that is a resource.

If `Worker`s are used, each thread has an independent `async_hooks` interface, and each thread will use a new set of async IDs.

Overview

Following is a simple overview of the public API.

```
import async_hooks from 'async_hooks';

// Return the ID of the current execution context.
const eid = async_hooks.executionAsyncId();

// Return the ID of the handle responsible for triggering the callback of the
// current execution scope to call.
const tid = async_hooks.triggerAsyncId();

// Create a new AsyncHook instance. All of these callbacks are optional.
const asyncHook =
    async_hooks.createHook({ init, before, after, destroy, promiseResolve });

// Allow callbacks of this AsyncHook instance to call. This is not an implicit
// action after running the constructor, and must be explicitly run to begin
// executing callbacks.
asyncHook.enable();

// Disable listening for new asynchronous events.
asyncHook.disable();

//
// The following are the callbacks that can be passed to createHook().
//
```

```
// init() is called during object construction. The resource may not have
// completed construction when this callback runs. Therefore, all fields of the
// resource referenced by "asyncId" may not have been populated.
function init(asyncId, type, triggerAsyncId, resource) { }

// before() is called just before the resource's callback is called. It can be
// called 0-N times for handles (such as TCPWrap), and will be called exactly 1
// time for requests (such as FSReqCallback).
function before(asyncId) { }

// after() is called just after the resource's callback has finished.
function after(asyncId) { }

// destroy() is called when the resource is destroyed.
function destroy(asyncId) { }

// promiseResolve() is called only for promise resources, when the
// resolve() function passed to the Promise constructor is invoked
// (either directly or through other means of resolving a promise).
function promiseResolve(asyncId) { }
```

```
const async_hooks = require('async_hooks');

// Return the ID of the current execution context.
const eid = async_hooks.executionAsyncId();

// Return the ID of the handle responsible for triggering the callback of the
// current execution scope to call.
const tid = async_hooks.triggerAsyncId();

// Create a new AsyncHook instance. All of these callbacks are optional.
const asyncHook =
    async_hooks.createHook({ init, before, after, destroy, promiseResolve });

// Allow callbacks of this AsyncHook instance to call. This is not an implicit
// action after running the constructor, and must be explicitly run to begin
// executing callbacks.
asyncHook.enable();

// Disable listening for new asynchronous events.
asyncHook.disable();

//
// The following are the callbacks that can be passed to createHook().
//

// init() is called during object construction. The resource may not have
// completed construction when this callback runs. Therefore, all fields of the
// resource referenced by "asyncId" may not have been populated.
function init(asyncId, type, triggerAsyncId, resource) { }
```

```
// before() is called just before the resource's callback is called. It can be
// called 0-N times for handles (such as TCPWrap), and will be called exactly 1
// time for requests (such as FSReqCallback).
function before(asyncId) { }

// after() is called just after the resource's callback has finished.
function after(asyncId) { }

// destroy() is called when the resource is destroyed.
function destroy(asyncId) { }

// promiseResolve() is called only for promise resources, when the
// resolve() function passed to the Promise constructor is invoked
// (either directly or through other means of resolving a promise).
function promiseResolve(asyncId) { }
```

`async_hooks.createHook(callbacks)`

- `callbacks` {Object} The Hook Callbacks to register
  - `init` {Function} The `init` callback.
  - `before` {Function} The `before` callback.
  - `after` {Function} The `after` callback.
  - `destroy` {Function} The `destroy` callback.
  - `promiseResolve` {Function} The `promiseResolve` callback.
- Returns: {AsyncHook} Instance used for disabling and enabling hooks

Registers functions to be called for different lifetime events of each async operation.

The callbacks `init()` / `before()` / `after()` / `destroy()` are called for the respective asynchronous event during a resource's lifetime.

All callbacks are optional. For example, if only resource cleanup needs to be tracked, then only the `destroy` callback needs to be passed. The specifics of all functions that can be passed to `callbacks` is in the Hook Callbacks section.

```
import { createHook } from 'async_hooks';

const asyncHook = createHook({
  init(asyncId, type, triggerAsyncId, resource) { },
  destroy(asyncId) { }
});
```

```
const async_hooks = require('async_hooks');

const asyncHook = async_hooks.createHook({
  init(asyncId, type, triggerAsyncId, resource) { },
  destroy(asyncId) { }
});
```

The callbacks will be inherited via the prototype chain:

```
class MyAsyncCallbacks {
  init(asyncId, type, triggerAsyncId, resource) { }
  destroy(asyncId) {}
}

class MyAddedCallbacks extends MyAsyncCallbacks {
  before(asyncId) { }
  after(asyncId) { }
}

const asyncHook = async_hooks.createHook(new MyAddedCallbacks());
```

Because promises are asynchronous resources whose lifecycle is tracked via the async hooks mechanism, the `init()` , `before()` , `after()` , and `destroy()` callbacks *must not* be async functions that return promises.

Error handling

If any `AsyncHook` callbacks throw, the application will print the stack trace and exit. The exit path does follow that of an uncaught exception, but all `'uncaughtException'` listeners are removed, thus forcing the process to exit. The `'exit'` callbacks will still be called unless the application is run with `--abort-on-uncaught-exception` , in which case a stack trace will be printed and the application exits, leaving a core file.

The reason for this error handling behavior is that these callbacks are running at potentially volatile points in an object's lifetime, for example during class construction and destruction. Because of this, it is deemed necessary to bring down the process quickly in order to prevent an unintentional abort in the future. This is subject to change in the future if a comprehensive analysis is performed to ensure an exception can follow the normal control flow without unintentional side effects.

Printing in `AsyncHook` callbacks

Because printing to the console is an asynchronous operation, `console.log()` will cause `AsyncHook` callbacks to be called. Using `console.log()` or similar asynchronous operations inside an `AsyncHook` callback function will cause an infinite recursion. An easy solution to this when debugging is to use a synchronous logging operation such as `fs.writeFileSync(file, msg, flag)` . This will print to the file and will not invoke `AsyncHook` recursively because it is synchronous.

```
import { writeFileSync } from 'fs';
import { format } from 'util';

function debug(...args) {
  // Use a function like this one when debugging inside an AsyncHook callback
  writeFileSync('log.out', `${format(...args)}\n`, { flag: 'a' });
}
```

```
const fs = require('fs');
const util = require('util');

function debug(...args) {
  // Use a function like this one when debugging inside an AsyncHook callback
  fs.writeFileSync('log.out', `${util.format(...args)}\n`, { flag: 'a' });
}
```

If an asynchronous operation is needed for logging, it is possible to keep track of what caused the asynchronous operation using the information provided by `AsyncHook` itself. The logging should then be skipped when it was the logging itself that caused the `AsyncHook` callback to be called. By doing this, the otherwise infinite recursion is broken.

Class: `AsyncHook`

The class `AsyncHook` exposes an interface for tracking lifetime events of asynchronous operations.

`asyncHook.enable()`

- Returns: {AsyncHook} A reference to `asyncHook`.

Enable the callbacks for a given `AsyncHook` instance. If no callbacks are provided, enabling is a no-op.

The `AsyncHook` instance is disabled by default. If the `AsyncHook` instance should be enabled immediately after creation, the following pattern can be used.

```
import { createHook } from 'async_hooks';

const hook = createHook(callbacks).enable();
```

```
const async_hooks = require('async_hooks');

const hook = async_hooks.createHook(callbacks).enable();
```

`asyncHook.disable()`

- Returns: {AsyncHook} A reference to `asyncHook`.

Disable the callbacks for a given `AsyncHook` instance from the global pool of `AsyncHook` callbacks to be executed. Once a hook has been disabled it will not be called again until enabled.

For API consistency `disable()` also returns the `AsyncHook` instance.

Hook callbacks

Key events in the lifetime of asynchronous events have been categorized into four areas: instantiation, before/after the callback is called, and when the instance is destroyed.

`init(asyncId, type, triggerAsyncId, resource)`

- `asyncId` {number} A unique ID for the async resource.
- `type` {string} The type of the async resource.
- `triggerAsyncId` {number} The unique ID of the async resource in whose execution context this async resource was created.
- `resource` {Object} Reference to the resource representing the async operation, needs to be released during *destroy*.

Called when a class is constructed that has the *possibility* to emit an asynchronous event. This *does not* mean the instance must call `before` / `after` before `destroy` is called, only that the possibility exists.

This behavior can be observed by doing something like opening a resource then closing it before the resource can be used. The following snippet demonstrates this.

```
import { createServer } from 'net';

createServer().listen(function() { this.close(); });
// OR
clearTimeout(setTimeout(() => {}, 10));
```

```
require('net').createServer().listen(function() { this.close(); });
// OR
clearTimeout(setTimeout(() => {}, 10));
```

Every new resource is assigned an ID that is unique within the scope of the current Node.js instance.

`type`

The `type` is a string identifying the type of resource that caused `init` to be called. Generally, it will correspond to the name of the resource's constructor.

Valid values are:

```
FSEVENTWRAP, FSREQCALLBACK, GETADDRINFOREQWRAP, GETNAMEINFOREQWRAP,
HTTPINCOMINGMESSAGE,
HTTPCLIENTREQUEST, JSSTREAM, PIPECONNECTWRAP, PIPEWRAP, PROCESSWRAP, QUERYWRAP,
SHUTDOWNWRAP, SIGNALWRAP, STATWATCHER, TCPCONNECTWRAP, TCPSERVERWRAP, TCPWRAP,
TTYWRAP, UDPSENDWRAP, UDPWRAP, WRITEWRAP, ZLIB, SSLCONNECTION, PBKDF2REQUEST,
RANDOMBYTESREQUEST, TLSWRAP, Microtask, Timeout, Immediate, TickObject
```

These values can change in any Node.js release. Furthermore users of `AsyncResource` likely provide other values.

There is also the `PROMISE` resource type, which is used to track `Promise` instances and asynchronous work scheduled by them.

Users are able to define their own `type` when using the public embedder API.

It is possible to have type name collisions. Embedders are encouraged to use unique prefixes, such as the npm package name, to prevent collisions when listening to the hooks.

`triggerAsyncId`

`triggerAsyncId` is the `asyncId` of the resource that caused (or "triggered") the new resource to initialize and that caused `init` to call. This is different from `async_hooks.executionAsyncId()` that only shows *when* a resource was created, while `triggerAsyncId` shows *why* a resource was created.

The following is a simple demonstration of `triggerAsyncId` :

```
import { createHook, executionAsyncId } from 'async_hooks';
import { stdout } from 'process';
import net from 'net';

createHook({
  init(asyncId, type, triggerAsyncId) {
    const eid = executionAsyncId();
    fs.writeSync(
      stdout.fd,
      `${type}(${asyncId}): trigger: ${triggerAsyncId} execution: ${eid}\n`);
  }
}).enable();

net.createServer((conn) => {}).listen(8080);
```

```
const { createHook, executionAsyncId } = require('async_hooks');
const { stdout } = require('process');
const net = require('net');

createHook({
  init(asyncId, type, triggerAsyncId) {
    const eid = executionAsyncId();
    fs.writeSync(
      stdout.fd,
      `${type}(${asyncId}): trigger: ${triggerAsyncId} execution: ${eid}\n`);
  }
}).enable();

net.createServer((conn) => {}).listen(8080);
```

Output when hitting the server with `nc localhost 8080` :

```
TCPSERVERWRAP(5): trigger: 1 execution: 1
TCPWRAP(7): trigger: 5 execution: 0
```

The `TCPSERVERWRAP` is the server which receives the connections.

The `TCPWRAP` is the new connection from the client. When a new connection is made, the `TCPWrap` instance is immediately constructed. This happens outside of any JavaScript stack. (An `executionAsyncId()` of `0` means that it is being executed from C++ with no JavaScript stack above it.) With only that information, it would be impossible to link resources together in terms of what caused them to be created, so `triggerAsyncId` is given the task of propagating what resource is responsible for the new resource's existence.

`resource`

`resource` is an object that represents the actual async resource that has been initialized. This can contain useful information that can vary based on the value of `type` . For instance, for the `GETADDRINFOREQWRAP` resource type, `resource` provides the host name used when looking up the IP address for the host in `net.Server.listen()` . The API for accessing this information is not supported, but using the Embedder API, users can provide and document their own resource objects. For example, such a resource object could contain the SQL query being executed.

In some cases the resource object is reused for performance reasons, it is thus not safe to use it as a key in a `WeakMap` or add properties to it.

Asynchronous context example

The following is an example with additional information about the calls to `init` between the `before` and `after` calls, specifically what the callback to `listen()` will look like. The output formatting is slightly more elaborate to make calling context easier to see.

```
const async_hooks = require('async_hooks');
const fs = require('fs');
const net = require('net');
const { fd } = process.stdout;

let indent = 0;
async_hooks.createHook({
  init(asyncId, type, triggerAsyncId) {
    const eid = async_hooks.executionAsyncId();
    const indentStr = ' '.repeat(indent);
    fs.writeSync(
      fd,
      `${indentStr}${type}(${asyncId}):` +
      ` trigger: ${triggerAsyncId} execution: ${eid}\n`);
  },
  before(asyncId) {
    const indentStr = ' '.repeat(indent);
    fs.writeSync(fd, `${indentStr}before:  ${asyncId}\n`);
    indent += 2;
  },
  after(asyncId) {
    indent -= 2;
    const indentStr = ' '.repeat(indent);
    fs.writeSync(fd, `${indentStr}after:  ${asyncId}\n`);
  },
  destroy(asyncId) {
    const indentStr = ' '.repeat(indent);
    fs.writeSync(fd, `${indentStr}destroy:  ${asyncId}\n`);
  },
}).enable();

net.createServer(() => {}).listen(8080, () => {
  // Let's wait 10ms before logging the server started.
  setTimeout(() => {
    console.log('>>>', async_hooks.executionAsyncId());
  }, 10);
});
```

Output from only starting the server:

```
TCPSERVERWRAP(5): trigger: 1 execution: 1
TickObject(6): trigger: 5 execution: 1
before:  6
```

```
  Timeout(7): trigger: 6 execution: 6
after:   6
destroy: 6
before:  7
>>> 7
  TickObject(8): trigger: 7 execution: 7
after:   7
before:  8
after:   8
```

As illustrated in the example, `executionAsyncId()` and `execution` each specify the value of the current execution context; which is delineated by calls to `before` and `after`.

Only using `execution` to graph resource allocation results in the following:

```
  root(1)
     ^
     |
TickObject(6)
     ^
     |
 Timeout(7)
```

The `TCPSERVERWRAP` is not part of this graph, even though it was the reason for `console.log()` being called. This is because binding to a port without a host name is a *synchronous* operation, but to maintain a completely asynchronous API the user's callback is placed in a `process.nextTick()`. Which is why `TickObject` is present in the output and is a 'parent' for `.listen()` callback.

The graph only shows *when* a resource was created, not *why*, so to track the *why* use `triggerAsyncId`. Which can be represented with the following graph:

```
 bootstrap(1)
     |
     v
TCPSERVERWRAP(5)
     |
     v
 TickObject(6)
     |
     v
  Timeout(7)
```

`before(asyncId)`

- `asyncId` {number}

When an asynchronous operation is initiated (such as a TCP server receiving a new connection) or completes (such as writing data to disk) a callback is called to notify the user. The `before` callback is called just before said callback is executed. `asyncId` is the unique identifier assigned to the resource about to execute the callback.

The `before` callback will be called 0 to N times. The `before` callback will typically be called 0 times if the asynchronous operation was cancelled or, for example, if no connections are received by a TCP server. Persistent asynchronous resources like a TCP server will typically call the `before` callback multiple times, while other operations like `fs.open()` will call it only once.

`after(asyncId)`

- `asyncId` {number}

Called immediately after the callback specified in `before` is completed.

If an uncaught exception occurs during execution of the callback, then `after` will run *after* the `'uncaughtException'` event is emitted or a `domain`'s handler runs.

`destroy(asyncId)`

- `asyncId` {number}

Called after the resource corresponding to `asyncId` is destroyed. It is also called asynchronously from the embedder API `emitDestroy()`.

Some resources depend on garbage collection for cleanup, so if a reference is made to the `resource` object passed to `init` it is possible that `destroy` will never be called, causing a memory leak in the application. If the resource does not depend on garbage collection, then this will not be an issue.

`promiseResolve(asyncId)`

- `asyncId` {number}

Called when the `resolve` function passed to the `Promise` constructor is invoked (either directly or through other means of resolving a promise).

`resolve()` does not do any observable synchronous work.

The `Promise` is not necessarily fulfilled or rejected at this point if the `Promise` was resolved by assuming the state of another `Promise`.

```js
new Promise((resolve) => resolve(true)).then((a) => {});
```

calls the following callbacks:

```
init for PROMISE with id 5, trigger id: 1
  promise resolve 5      # corresponds to resolve(true)
init for PROMISE with id 6, trigger id: 5  # the Promise returned by then()
  before 6               # the then() callback is entered
  promise resolve 6      # the then() callback resolves the promise by returning
  after 6
```

`async_hooks.executionAsyncResource()`

- Returns: {Object} The resource representing the current execution. Useful to store data within the resource.

Resource objects returned by `executionAsyncResource()` are most often internal Node.js handle objects with undocumented APIs. Using any functions or properties on the object is likely to crash your application and should be avoided.

Using `executionAsyncResource()` in the top-level execution context will return an empty object as there is no handle or request object to use, but having an object representing the top-level can be helpful.

```
import { open } from 'fs';
import { executionAsyncId, executionAsyncResource } from 'async_hooks';

console.log(executionAsyncId(), executionAsyncResource());  // 1 {}
open(new URL(import.meta.url), 'r', (err, fd) => {
  console.log(executionAsyncId(), executionAsyncResource());  // 7 FSReqWrap
});
```

```
const { open } = require('fs');
const { executionAsyncId, executionAsyncResource } = require('async_hooks');

console.log(executionAsyncId(), executionAsyncResource());  // 1 {}
open(__filename, 'r', (err, fd) => {
  console.log(executionAsyncId(), executionAsyncResource());  // 7 FSReqWrap
});
```

This can be used to implement continuation local storage without the use of a tracking `Map` to store the metadata:

```
import { createServer } from 'http';
import {
  executionAsyncId,
  executionAsyncResource,
  createHook
} from 'async_hooks';
const sym = Symbol('state'); // Private symbol to avoid pollution

createHook({
  init(asyncId, type, triggerAsyncId, resource) {
    const cr = executionAsyncResource();
    if (cr) {
      resource[sym] = cr[sym];
    }
  }
}).enable();

const server = createServer((req, res) => {
  executionAsyncResource()[sym] = { state: req.url };
  setTimeout(function() {
    res.end(JSON.stringify(executionAsyncResource()[sym]));
  }, 100);
}).listen(3000);
```

```
const { createServer } = require('http');
const {
  executionAsyncId,
  executionAsyncResource,
  createHook
} = require('async_hooks');
const sym = Symbol('state'); // Private symbol to avoid pollution

createHook({
  init(asyncId, type, triggerAsyncId, resource) {
    const cr = executionAsyncResource();
    if (cr) {
      resource[sym] = cr[sym];
    }
  }
}).enable();

const server = createServer((req, res) => {
  executionAsyncResource()[sym] = { state: req.url };
  setTimeout(function() {
    res.end(JSON.stringify(executionAsyncResource()[sym]));
  }, 100);
}).listen(3000);
```

`async_hooks.executionAsyncId()`

- Returns: {number} The `asyncId` of the current execution context. Useful to track when something calls.

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

console.log(executionAsyncId());  // 1 - bootstrap
fs.open(path, 'r', (err, fd) => {
  console.log(executionAsyncId());  // 6 - open()
});
```

```
const async_hooks = require('async_hooks');

console.log(async_hooks.executionAsyncId());  // 1 - bootstrap
fs.open(path, 'r', (err, fd) => {
  console.log(async_hooks.executionAsyncId());  // 6 - open()
});
```

The ID returned from `executionAsyncId()` is related to execution timing, not causality (which is covered by `triggerAsyncId()`):

```
const server = net.createServer((conn) => {
  // Returns the ID of the server, not of the new connection, because the
  // callback runs in the execution scope of the server's MakeCallback().
  async_hooks.executionAsyncId();

}).listen(port, () => {
  // Returns the ID of a TickObject (process.nextTick()) because all
  // callbacks passed to .listen() are wrapped in a nextTick().
  async_hooks.executionAsyncId();
});
```

Promise contexts may not get precise `executionAsyncIds` by default. See the section on [promise execution tracking](#promise-execution-tracking).

`async_hooks.triggerAsyncId()`

- Returns: {number} The ID of the resource responsible for calling the callback that is currently being executed.

```
const server = net.createServer((conn) => {
  // The resource that caused (or triggered) this callback to be called
  // was that of the new connection. Thus the return value of triggerAsyncId()
  // is the asyncId of "conn".
  async_hooks.triggerAsyncId();

}).listen(port, () => {
  // Even though all callbacks passed to .listen() are wrapped in a nextTick()
  // the callback itself exists because the call to the server's .listen()
  // was made. So the return value would be the ID of the server.
  async_hooks.triggerAsyncId();
});
```

Promise contexts may not get valid `triggerAsyncId`s by default. See the section on [promise execution tracking](#promise-execution-tracking).

`async_hooks.asyncWrapProviders`

- Returns: A map of provider types to the corresponding numeric id. This map contains all the event types that might be emitted by the `async_hooks.init()` event.

This feature suppresses the deprecated usage of `process.binding('async_wrap').Providers`. See: [DEP0111](#)

Promise execution tracking

By default, promise executions are not assigned `asyncId`s due to the relatively expensive nature of the [promise introspection API](#) provided by V8. This means that programs using promises or `async`/`await` will not get correct execution and trigger ids for promise callback contexts by default.

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

Promise.resolve(1729).then(() => {
  console.log(`eid ${executionAsyncId()} tid ${triggerAsyncId()}`);
});
// produces:
// eid 1 tid 0
```

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

Promise.resolve(1729).then(() => {
  console.log(`eid ${executionAsyncId()} tid ${triggerAsyncId()}`);
});
// produces:
// eid 1 tid 0
```

Observe that the `then()` callback claims to have executed in the context of the outer scope even though there was an asynchronous hop involved. Also, the `triggerAsyncId` value is `0`, which means that we are missing context about the resource that caused (triggered) the `then()` callback to be executed.

Installing async hooks via `async_hooks.createHook` enables promise execution tracking:

```
import { createHook, executionAsyncId, triggerAsyncId } from 'async_hooks';
createHook({ init() {} }).enable(); // forces PromiseHooks to be enabled.
Promise.resolve(1729).then(() => {
  console.log(`eid ${executionAsyncId()} tid ${triggerAsyncId()}`);
});
// produces:
// eid 7 tid 6
```

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

createHook({ init() {} }).enable(); // forces PromiseHooks to be enabled.
Promise.resolve(1729).then(() => {
  console.log(`eid ${executionAsyncId()} tid ${triggerAsyncId()}`);
});
// produces:
// eid 7 tid 6
```

In this example, adding any actual hook function enabled the tracking of promises. There are two promises in the example above; the promise created by `Promise.resolve()` and the promise returned by the call to `then()`. In the example above, the first promise got the `asyncId` `6` and the latter got `asyncId` `7`. During the execution of the `then()` callback, we are executing in the context of promise with `asyncId` `7`. This promise was triggered by async resource `6`.

Another subtlety with promises is that `before` and `after` callbacks are run only on chained promises. That means promises not created by `then()`/`catch()` will not have the `before` and `after` callbacks fired on them. For more details see the details of the V8 PromiseHooks API.

JavaScript embedder API

Library developers that handle their own asynchronous resources performing tasks like I/O, connection pooling, or managing callback queues may use the `AsyncResource` JavaScript API so that all the appropriate callbacks are called.

Class: `AsyncResource`

The documentation for this class has moved `AsyncResource` .

Class: `AsyncLocalStorage`

The documentation for this class has moved `AsyncLocalStorage` .