/\*\*

\* The `async\_hooks` module provides an API to track asynchronous resources. It

\* can be accessed using:

\*

\* ```js

\* import async\_hooks from 'async\_hooks';

\* ```

\* @experimental

\* @see [source](https://github.com/nodejs/node/blob/v17.0.0/lib/async\_hooks.js)

\*/

declare module 'async\_hooks' {

/\*\*

\* ```js

\* import { executionAsyncId } from 'async\_hooks';

\*

\* console.log(executionAsyncId()); // 1 - bootstrap

\* fs.open(path, 'r', (err, fd) => {

\* console.log(executionAsyncId()); // 6 - open()

\* });

\* ```

\*

\* The ID returned from `executionAsyncId()` is related to execution timing, not

\* causality (which is covered by `triggerAsyncId()`):

\*

\* ```js

\* 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`.

\* @since v8.1.0

\* @return The `asyncId` of the current execution context. Useful to track when something calls.

\*/

function executionAsyncId(): number;

/\*\*

\* 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.

\*

\* ```js

\* 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

\* });

\* ```

\*

\* This can be used to implement continuation local storage without the

\* use of a tracking `Map` to store the metadata:

\*

\* ```js

\* 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);

\* ```

\* @since v13.9.0, v12.17.0

\* @return The resource representing the current execution. Useful to store data within the resource.

\*/

function executionAsyncResource(): object;

/\*\*

\* ```js

\* 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`.

\* @return The ID of the resource responsible for calling the callback that is currently being executed.

\*/

function triggerAsyncId(): number;

interface HookCallbacks {

/\*\*

\* Called when a class is constructed that has the possibility to emit an asynchronous event.

\* @param asyncId a unique ID for the async resource

\* @param type the type of the async resource

\* @param triggerAsyncId the unique ID of the async resource in whose execution context this async resource was created

\* @param resource reference to the resource representing the async operation, needs to be released during destroy

\*/

init?(asyncId: number, type: string, triggerAsyncId: number, resource: object): void;

/\*\*

\* When an asynchronous operation is initiated or completes a callback is called to notify the user.

\* The before callback is called just before said callback is executed.

\* @param asyncId the unique identifier assigned to the resource about to execute the callback.

\*/

before?(asyncId: number): void;

/\*\*

\* Called immediately after the callback specified in before is completed.

\* @param asyncId the unique identifier assigned to the resource which has executed the callback.

\*/

after?(asyncId: number): void;

/\*\*

\* Called when a promise has resolve() called. This may not be in the same execution id

\* as the promise itself.

\* @param asyncId the unique id for the promise that was resolve()d.

\*/

promiseResolve?(asyncId: number): void;

/\*\*

\* Called after the resource corresponding to asyncId is destroyed

\* @param asyncId a unique ID for the async resource

\*/

destroy?(asyncId: number): void;

}

interface AsyncHook {

/\*\*

\* Enable the callbacks for a given AsyncHook instance. If no callbacks are provided enabling is a noop.

\*/

enable(): this;

/\*\*

\* 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.

\*/

disable(): this;

}

/\*\*

\* 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.

\*

\* ```js

\* import { createHook } from 'async\_hooks';

\*

\* const asyncHook = createHook({

\* init(asyncId, type, triggerAsyncId, resource) { },

\* destroy(asyncId) { }

\* });

\* ```

\*

\* The callbacks will be inherited via the prototype chain:

\*

\* ```js

\* 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.

\* @since v8.1.0

\* @param callbacks The `Hook Callbacks` to register

\* @return Instance used for disabling and enabling hooks

\*/

function createHook(callbacks: HookCallbacks): AsyncHook;

interface AsyncResourceOptions {

/\*\*

\* The ID of the execution context that created this async event.

\* @default executionAsyncId()

\*/

triggerAsyncId?: number | undefined;

/\*\*

\* Disables automatic `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.

\* @default false

\*/

requireManualDestroy?: boolean | undefined;

}

/\*\*

\* 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.

\*

\* ```js

\* 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();

\* ```

\*/

class AsyncResource {

/\*\*

\* AsyncResource() is meant to be extended. Instantiating a

\* new AsyncResource() also triggers init. If triggerAsyncId is omitted then

\* async\_hook.executionAsyncId() is used.

\* @param type The type of async event.

\* @param triggerAsyncId The ID of the execution context that created

\* this async event (default: `executionAsyncId()`), or an

\* AsyncResourceOptions object (since 9.3)

\*/

constructor(type: string, triggerAsyncId?: number | AsyncResourceOptions);

/\*\*

\* 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.

\* @since v14.8.0, v12.19.0

\* @param fn The function to bind to the current execution context.

\* @param type An optional name to associate with the underlying `AsyncResource`.

\*/

static bind<Func extends (this: ThisArg, ...args: any[]) => any, ThisArg>(

fn: Func,

type?: string,

thisArg?: ThisArg

): Func & {

asyncResource: AsyncResource;

};

/\*\*

\* 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.

\* @since v14.8.0, v12.19.0

\* @param fn The function to bind to the current `AsyncResource`.

\*/

bind<Func extends (...args: any[]) => any>(

fn: Func

): Func & {

asyncResource: AsyncResource;

};

/\*\*

\* 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.

\* @since v9.6.0

\* @param fn The function to call in the execution context of this async resource.

\* @param thisArg The receiver to be used for the function call.

\* @param args Optional arguments to pass to the function.

\*/

runInAsyncScope<This, Result>(fn: (this: This, ...args: any[]) => Result, thisArg?: This, ...args: any[]): Result;

/\*\*

\* 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.

\* @return A reference to `asyncResource`.

\*/

emitDestroy(): this;

/\*\*

\* @return The unique `asyncId` assigned to the resource.

\*/

asyncId(): number;

/\*\*

\*

\* @return The same `triggerAsyncId` that is passed to the `AsyncResource` constructor.

\*/

triggerAsyncId(): number;

}

/\*\*

\* 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.

\*

\* ```js

\* 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

\* ```

\*

\* Each instance of `AsyncLocalStorage` maintains an independent storage context.

\* Multiple instances can safely exist simultaneously without risk of interfering

\* with each other data.

\* @since v13.10.0, v12.17.0

\*/

class AsyncLocalStorage<T> {

/\*\*

\* 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.

\* @since v13.10.0, v12.17.0

\* @experimental

\*/

disable(): void;

/\*\*

\* Returns the current store.

\* If called outside of an asynchronous context initialized by

\* calling `asyncLocalStorage.run()` or `asyncLocalStorage.enterWith()`, it

\* returns `undefined`.

\* @since v13.10.0, v12.17.0

\*/

getStore(): T | undefined;

/\*\*

\* 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:

\*

\* ```js

\* 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

\* }

\* ```

\* @since v13.10.0, v12.17.0

\*/

run<R, TArgs extends any[]>(store: T, callback: (...args: TArgs) => R, ...args: TArgs): R;

/\*\*

\* 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:

\*

\* ```js

\* // 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

\* }

\* ```

\* @since v13.10.0, v12.17.0

\* @experimental

\*/

exit<R, TArgs extends any[]>(callback: (...args: TArgs) => R, ...args: TArgs): R;

/\*\*

\* Transitions into the context for the remainder of the current

\* synchronous execution and then persists the store through any following

\* asynchronous calls.

\*

\* Example:

\*

\* ```js

\* 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.

\*

\* ```js

\* 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

\* ```

\* @since v13.11.0, v12.17.0

\* @experimental

\*/

enterWith(store: T): void;

}

}

declare module 'node:async\_hooks' {

export \* from 'async\_hooks';

}