/\*\*

\* \*\*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.

\* @deprecated Since v1.4.2 - Deprecated

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

\*/

declare module 'domain' {

import EventEmitter = require('node:events');

/\*\*

\* 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.

\*/

class Domain extends EventEmitter {

/\*\*

\* An array of timers and event emitters that have been explicitly added

\* to the domain.

\*/

members: Array<EventEmitter | NodeJS.Timer>;

/\*\*

\* 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 {@link 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.

\*/

enter(): void;

/\*\*

\* 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.

\*/

exit(): void;

/\*\*

\* 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.

\*

\* ```js

\* 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.

\*/

run<T>(fn: (...args: any[]) => T, ...args: any[]): T;

/\*\*

\* 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.

\* @param emitter emitter or timer to be added to the domain

\*/

add(emitter: EventEmitter | NodeJS.Timer): void;

/\*\*

\* The opposite of {@link add}. Removes domain handling from the

\* specified emitter.

\* @param emitter emitter or timer to be removed from the domain

\*/

remove(emitter: EventEmitter | NodeJS.Timer): void;

/\*\*

\* 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.

\*

\* ```js

\* 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.

\* });

\* ```

\* @param callback The callback function

\* @return The bound function

\*/

bind<T extends Function>(callback: T): T;

/\*\*

\* This method is almost identical to {@link bind}. 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.

\*

\* ```js

\* 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.

\* });

\* ```

\* @param callback The callback function

\* @return The intercepted function

\*/

intercept<T extends Function>(callback: T): T;

}

function create(): Domain;

}

declare module 'node:domain' {

export \* from 'domain';

}