/\*\*

\* The `vm` module enables compiling and running code within V8 Virtual

\* Machine contexts. \*\*The `vm` module is not a security mechanism. Do\*\*

\* \*\*not use it to run untrusted code.\*\*

\*

\* JavaScript code can be compiled and run immediately or

\* compiled, saved, and run later.

\*

\* A common use case is to run the code in a different V8 Context. This means

\* invoked code has a different global object than the invoking code.

\*

\* One can provide the context by `contextifying` an

\* object. The invoked code treats any property in the context like a

\* global variable. Any changes to global variables caused by the invoked

\* code are reflected in the context object.

\*

\* ```js

\* const vm = require('vm');

\*

\* const x = 1;

\*

\* const context = { x: 2 };

\* vm.createContext(context); // Contextify the object.

\*

\* const code = 'x += 40; var y = 17;';

\* // `x` and `y` are global variables in the context.

\* // Initially, x has the value 2 because that is the value of context.x.

\* vm.runInContext(code, context);

\*

\* console.log(context.x); // 42

\* console.log(context.y); // 17

\*

\* console.log(x); // 1; y is not defined.

\* ```

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

\*/

declare module 'vm' {

interface Context extends NodeJS.Dict<any> {}

interface BaseOptions {

/\*\*

\* Specifies the filename used in stack traces produced by this script.

\* Default: `''`.

\*/

filename?: string | undefined;

/\*\*

\* Specifies the line number offset that is displayed in stack traces produced by this script.

\* Default: `0`.

\*/

lineOffset?: number | undefined;

/\*\*

\* Specifies the column number offset that is displayed in stack traces produced by this script.

\* @default 0

\*/

columnOffset?: number | undefined;

}

interface ScriptOptions extends BaseOptions {

displayErrors?: boolean | undefined;

timeout?: number | undefined;

cachedData?: Buffer | undefined;

/\*\* @deprecated in favor of `script.createCachedData()` \*/

produceCachedData?: boolean | undefined;

}

interface RunningScriptOptions extends BaseOptions {

/\*\*

\* When `true`, if an `Error` occurs while compiling the `code`, the line of code causing the error is attached to the stack trace.

\* Default: `true`.

\*/

displayErrors?: boolean | undefined;

/\*\*

\* Specifies the number of milliseconds to execute code before terminating execution.

\* If execution is terminated, an `Error` will be thrown. This value must be a strictly positive integer.

\*/

timeout?: number | undefined;

/\*\*

\* If `true`, the execution will be terminated when `SIGINT` (Ctrl+C) is received.

\* Existing handlers for the event that have been attached via `process.on('SIGINT')` will be disabled during script execution, but will continue to work after that.

\* If execution is terminated, an `Error` will be thrown.

\* Default: `false`.

\*/

breakOnSigint?: boolean | undefined;

/\*\*

\* If set to `afterEvaluate`, microtasks will be run immediately after the script has run.

\*/

microtaskMode?: 'afterEvaluate' | undefined;

}

interface CompileFunctionOptions extends BaseOptions {

/\*\*

\* Provides an optional data with V8's code cache data for the supplied source.

\*/

cachedData?: Buffer | undefined;

/\*\*

\* Specifies whether to produce new cache data.

\* Default: `false`,

\*/

produceCachedData?: boolean | undefined;

/\*\*

\* The sandbox/context in which the said function should be compiled in.

\*/

parsingContext?: Context | undefined;

/\*\*

\* An array containing a collection of context extensions (objects wrapping the current scope) to be applied while compiling

\*/

contextExtensions?: Object[] | undefined;

}

interface CreateContextOptions {

/\*\*

\* Human-readable name of the newly created context.

\* @default 'VM Context i' Where i is an ascending numerical index of the created context.

\*/

name?: string | undefined;

/\*\*

\* Corresponds to the newly created context for display purposes.

\* The origin should be formatted like a `URL`, but with only the scheme, host, and port (if necessary),

\* like the value of the `url.origin` property of a URL object.

\* Most notably, this string should omit the trailing slash, as that denotes a path.

\* @default ''

\*/

origin?: string | undefined;

codeGeneration?:

| {

/\*\*

\* If set to false any calls to eval or function constructors (Function, GeneratorFunction, etc)

\* will throw an EvalError.

\* @default true

\*/

strings?: boolean | undefined;

/\*\*

\* If set to false any attempt to compile a WebAssembly module will throw a WebAssembly.CompileError.

\* @default true

\*/

wasm?: boolean | undefined;

}

| undefined;

/\*\*

\* If set to `afterEvaluate`, microtasks will be run immediately after the script has run.

\*/

microtaskMode?: 'afterEvaluate' | undefined;

}

type MeasureMemoryMode = 'summary' | 'detailed';

interface MeasureMemoryOptions {

/\*\*

\* @default 'summary'

\*/

mode?: MeasureMemoryMode | undefined;

context?: Context | undefined;

}

interface MemoryMeasurement {

total: {

jsMemoryEstimate: number;

jsMemoryRange: [number, number];

};

}

/\*\*

\* Instances of the `vm.Script` class contain precompiled scripts that can be

\* executed in specific contexts.

\* @since v0.3.1

\*/

class Script {

constructor(code: string, options?: ScriptOptions);

/\*\*

\* Runs the compiled code contained by the `vm.Script` object within the given`contextifiedObject` and returns the result. Running code does not have access

\* to local scope.

\*

\* The following example compiles code that increments a global variable, sets

\* the value of another global variable, then execute the code multiple times.

\* The globals are contained in the `context` object.

\*

\* ```js

\* const vm = require('vm');

\*

\* const context = {

\* animal: 'cat',

\* count: 2

\* };

\*

\* const script = new vm.Script('count += 1; name = "kitty";');

\*

\* vm.createContext(context);

\* for (let i = 0; i < 10; ++i) {

\* script.runInContext(context);

\* }

\*

\* console.log(context);

\* // Prints: { animal: 'cat', count: 12, name: 'kitty' }

\* ```

\*

\* Using the `timeout` or `breakOnSigint` options will result in new event loops

\* and corresponding threads being started, which have a non-zero performance

\* overhead.

\* @since v0.3.1

\* @param contextifiedObject A `contextified` object as returned by the `vm.createContext()` method.

\* @return the result of the very last statement executed in the script.

\*/

runInContext(contextifiedObject: Context, options?: RunningScriptOptions): any;

/\*\*

\* First contextifies the given `contextObject`, runs the compiled code contained

\* by the `vm.Script` object within the created context, and returns the result.

\* Running code does not have access to local scope.

\*

\* The following example compiles code that sets a global variable, then executes

\* the code multiple times in different contexts. The globals are set on and

\* contained within each individual `context`.

\*

\* ```js

\* const vm = require('vm');

\*

\* const script = new vm.Script('globalVar = "set"');

\*

\* const contexts = [{}, {}, {}];

\* contexts.forEach((context) => {

\* script.runInNewContext(context);

\* });

\*

\* console.log(contexts);

\* // Prints: [{ globalVar: 'set' }, { globalVar: 'set' }, { globalVar: 'set' }]

\* ```

\* @since v0.3.1

\* @param contextObject An object that will be `contextified`. If `undefined`, a new object will be created.

\* @return the result of the very last statement executed in the script.

\*/

runInNewContext(contextObject?: Context, options?: RunningScriptOptions): any;

/\*\*

\* Runs the compiled code contained by the `vm.Script` within the context of the

\* current `global` object. Running code does not have access to local scope, but\_does\_ have access to the current `global` object.

\*

\* The following example compiles code that increments a `global` variable then

\* executes that code multiple times:

\*

\* ```js

\* const vm = require('vm');

\*

\* global.globalVar = 0;

\*

\* const script = new vm.Script('globalVar += 1', { filename: 'myfile.vm' });

\*

\* for (let i = 0; i < 1000; ++i) {

\* script.runInThisContext();

\* }

\*

\* console.log(globalVar);

\*

\* // 1000

\* ```

\* @since v0.3.1

\* @return the result of the very last statement executed in the script.

\*/

runInThisContext(options?: RunningScriptOptions): any;

/\*\*

\* Creates a code cache that can be used with the `Script` constructor's`cachedData` option. Returns a `Buffer`. This method may be called at any

\* time and any number of times.

\*

\* ```js

\* const script = new vm.Script(`

\* function add(a, b) {

\* return a + b;

\* }

\*

\* const x = add(1, 2);

\* `);

\*

\* const cacheWithoutX = script.createCachedData();

\*

\* script.runInThisContext();

\*

\* const cacheWithX = script.createCachedData();

\* ```

\* @since v10.6.0

\*/

createCachedData(): Buffer;

/\*\* @deprecated in favor of `script.createCachedData()` \*/

cachedDataProduced?: boolean | undefined;

cachedDataRejected?: boolean | undefined;

cachedData?: Buffer | undefined;

}

/\*\*

\* If given a `contextObject`, the `vm.createContext()` method will `prepare

\* that object` so that it can be used in calls to {@link runInContext} or `script.runInContext()`. Inside such scripts,

\* the `contextObject` will be the global object, retaining all of its existing

\* properties but also having the built-in objects and functions any standard [global object](https://es5.github.io/#x15.1) has. Outside of scripts run by the vm module, global variables

\* will remain unchanged.

\*

\* ```js

\* const vm = require('vm');

\*

\* global.globalVar = 3;

\*

\* const context = { globalVar: 1 };

\* vm.createContext(context);

\*

\* vm.runInContext('globalVar \*= 2;', context);

\*

\* console.log(context);

\* // Prints: { globalVar: 2 }

\*

\* console.log(global.globalVar);

\* // Prints: 3

\* ```

\*

\* If `contextObject` is omitted (or passed explicitly as `undefined`), a new,

\* empty `contextified` object will be returned.

\*

\* The `vm.createContext()` method is primarily useful for creating a single

\* context that can be used to run multiple scripts. For instance, if emulating a

\* web browser, the method can be used to create a single context representing a

\* window's global object, then run all `<script>` tags together within that

\* context.

\*

\* The provided `name` and `origin` of the context are made visible through the

\* Inspector API.

\* @since v0.3.1

\* @return contextified object.

\*/

function createContext(sandbox?: Context, options?: CreateContextOptions): Context;

/\*\*

\* Returns `true` if the given `object` object has been `contextified` using {@link createContext}.

\* @since v0.11.7

\*/

function isContext(sandbox: Context): boolean;

/\*\*

\* The `vm.runInContext()` method compiles `code`, runs it within the context of

\* the `contextifiedObject`, then returns the result. Running code does not have

\* access to the local scope. The `contextifiedObject` object \_must\_ have been

\* previously `contextified` using the {@link createContext} method.

\*

\* If `options` is a string, then it specifies the filename.

\*

\* The following example compiles and executes different scripts using a single `contextified` object:

\*

\* ```js

\* const vm = require('vm');

\*

\* const contextObject = { globalVar: 1 };

\* vm.createContext(contextObject);

\*

\* for (let i = 0; i < 10; ++i) {

\* vm.runInContext('globalVar \*= 2;', contextObject);

\* }

\* console.log(contextObject);

\* // Prints: { globalVar: 1024 }

\* ```

\* @since v0.3.1

\* @param code The JavaScript code to compile and run.

\* @param contextifiedObject The `contextified` object that will be used as the `global` when the `code` is compiled and run.

\* @return the result of the very last statement executed in the script.

\*/

function runInContext(code: string, contextifiedObject: Context, options?: RunningScriptOptions | string): any;

/\*\*

\* The `vm.runInNewContext()` first contextifies the given `contextObject` (or

\* creates a new `contextObject` if passed as `undefined`), compiles the `code`,

\* runs it within the created context, then returns the result. Running code

\* does not have access to the local scope.

\*

\* If `options` is a string, then it specifies the filename.

\*

\* The following example compiles and executes code that increments a global

\* variable and sets a new one. These globals are contained in the `contextObject`.

\*

\* ```js

\* const vm = require('vm');

\*

\* const contextObject = {

\* animal: 'cat',

\* count: 2

\* };

\*

\* vm.runInNewContext('count += 1; name = "kitty"', contextObject);

\* console.log(contextObject);

\* // Prints: { animal: 'cat', count: 3, name: 'kitty' }

\* ```

\* @since v0.3.1

\* @param code The JavaScript code to compile and run.

\* @param contextObject An object that will be `contextified`. If `undefined`, a new object will be created.

\* @return the result of the very last statement executed in the script.

\*/

function runInNewContext(code: string, contextObject?: Context, options?: RunningScriptOptions | string): any;

/\*\*

\* `vm.runInThisContext()` compiles `code`, runs it within the context of the

\* current `global` and returns the result. Running code does not have access to

\* local scope, but does have access to the current `global` object.

\*

\* If `options` is a string, then it specifies the filename.

\*

\* The following example illustrates using both `vm.runInThisContext()` and

\* the JavaScript [`eval()`](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\_Objects/eval) function to run the same code:

\*

\* ```js

\* const vm = require('vm');

\* let localVar = 'initial value';

\*

\* const vmResult = vm.runInThisContext('localVar = "vm";');

\* console.log(`vmResult: '${vmResult}', localVar: '${localVar}'`);

\* // Prints: vmResult: 'vm', localVar: 'initial value'

\*

\* const evalResult = eval('localVar = "eval";');

\* console.log(`evalResult: '${evalResult}', localVar: '${localVar}'`);

\* // Prints: evalResult: 'eval', localVar: 'eval'

\* ```

\*

\* Because `vm.runInThisContext()` does not have access to the local scope,`localVar` is unchanged. In contrast,

\* [`eval()`](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\_Objects/eval) \_does\_ have access to the

\* local scope, so the value `localVar` is changed. In this way`vm.runInThisContext()` is much like an [indirect `eval()` call](https://es5.github.io/#x10.4.2), e.g.`(0,eval)('code')`.

\*

\* ## Example: Running an HTTP server within a VM

\*

\* When using either `script.runInThisContext()` or {@link runInThisContext}, the code is executed within the current V8 global

\* context. The code passed to this VM context will have its own isolated scope.

\*

\* In order to run a simple web server using the `http` module the code passed to

\* the context must either call `require('http')` on its own, or have a reference

\* to the `http` module passed to it. For instance:

\*

\* ```js

\* 'use strict';

\* const vm = require('vm');

\*

\* const code = `

\* ((require) => {

\* const http = require('http');

\*

\* http.createServer((request, response) => {

\* response.writeHead(200, { 'Content-Type': 'text/plain' });

\* response.end('Hello World\\n');

\* }).listen(8124);

\*

\* console.log('Server running at http://127.0.0.1:8124/');

\* })`;

\*

\* vm.runInThisContext(code)(require);

\* ```

\*

\* The `require()` in the above case shares the state with the context it is

\* passed from. This may introduce risks when untrusted code is executed, e.g.

\* altering objects in the context in unwanted ways.

\* @since v0.3.1

\* @param code The JavaScript code to compile and run.

\* @return the result of the very last statement executed in the script.

\*/

function runInThisContext(code: string, options?: RunningScriptOptions | string): any;

/\*\*

\* Compiles the given code into the provided context (if no context is

\* supplied, the current context is used), and returns it wrapped inside a

\* function with the given `params`.

\* @since v10.10.0

\* @param code The body of the function to compile.

\* @param params An array of strings containing all parameters for the function.

\*/

function compileFunction(code: string, params?: ReadonlyArray<string>, options?: CompileFunctionOptions): Function;

/\*\*

\* Measure the memory known to V8 and used by all contexts known to the

\* current V8 isolate, or the main context.

\*

\* The format of the object that the returned Promise may resolve with is

\* specific to the V8 engine and may change from one version of V8 to the next.

\*

\* The returned result is different from the statistics returned by`v8.getHeapSpaceStatistics()` in that `vm.measureMemory()` measure the

\* memory reachable by each V8 specific contexts in the current instance of

\* the V8 engine, while the result of `v8.getHeapSpaceStatistics()` measure

\* the memory occupied by each heap space in the current V8 instance.

\*

\* ```js

\* const vm = require('vm');

\* // Measure the memory used by the main context.

\* vm.measureMemory({ mode: 'summary' })

\* // This is the same as vm.measureMemory()

\* .then((result) => {

\* // The current format is:

\* // {

\* // total: {

\* // jsMemoryEstimate: 2418479, jsMemoryRange: [ 2418479, 2745799 ]

\* // }

\* // }

\* console.log(result);

\* });

\*

\* const context = vm.createContext({ a: 1 });

\* vm.measureMemory({ mode: 'detailed', execution: 'eager' })

\* .then((result) => {

\* // Reference the context here so that it won't be GC'ed

\* // until the measurement is complete.

\* console.log(context.a);

\* // {

\* // total: {

\* // jsMemoryEstimate: 2574732,

\* // jsMemoryRange: [ 2574732, 2904372 ]

\* // },

\* // current: {

\* // jsMemoryEstimate: 2438996,

\* // jsMemoryRange: [ 2438996, 2768636 ]

\* // },

\* // other: [

\* // {

\* // jsMemoryEstimate: 135736,

\* // jsMemoryRange: [ 135736, 465376 ]

\* // }

\* // ]

\* // }

\* console.log(result);

\* });

\* ```

\* @since v13.10.0

\* @experimental

\*/

function measureMemory(options?: MeasureMemoryOptions): Promise<MemoryMeasurement>;

}

declare module 'node:vm' {

export \* from 'vm';

}