Node.js v16.20.0 documentation

VM (executing JavaScript)

Stability: 2 - Stable

Source Code: lib/vm.js

The `node:vm` module enables compiling and running code within V8 Virtual Machine contexts.

The `node: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.

```
const vm = require('node: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.
```

Class: vm.Script

Instances of the `vm.Script` class contain precompiled scripts that can be executed in specific contexts.

new vm.Script(code[, options])

- `code` `<string>` The JavaScript code to compile.
- `options` `<Object>` | `<string>`
  - `filename` `<string>` Specifies the filename used in stack traces produced by this script. **Default:** `'evalmachine.<anonymous>'`.
  - `lineOffset` `<number>` Specifies the line number offset that is displayed in stack traces produced by this script. **Default:** `0`.
  - `columnOffset` `<number>` Specifies the first-line column number offset that is displayed in stack traces produced by this script. **Default:** `0`.

- `cachedData` `<Buffer>` | `<TypedArray>` | `<DataView>` Provides an optional `Buffer` or `TypedArray`, or `DataView` with V8's code cache data for the supplied source. When supplied, the `cachedDataRejected` value will be set to either `true` or `false` depending on acceptance of the data by V8.
- `produceCachedData` `<boolean>` When `true` and no `cachedData` is present, V8 will attempt to produce code cache data for `code`. Upon success, a `Buffer` with V8's code cache data will be produced and stored in the `cachedData` property of the returned `vm.Script` instance. The `cachedDataProduced` value will be set to either `true` or `false` depending on whether code cache data is produced successfully. This option is **deprecated** in favor of `script.createCachedData()`. **Default:** `false`.
- `importModuleDynamically` `<Function>` Called during evaluation of this module when `import()` is called. If this option is not specified, calls to `import()` will reject with `ERR_VM_DYNAMIC_IMPORT_CALLBACK_MISSING`. This option is part of the experimental modules API. We do not recommend using it in a production environment.
  - `specifier` `<string>` specifier passed to `import()`
  - `script` `<vm.Script>`
  - `importAssertions` `<Object>` The `"assert"` value passed to the `optionsExpression` optional parameter, or an empty object if no value was provided.
  - Returns: `<Module Namespace Object>` | `<vm.Module>` Returning a `vm.Module` is recommended in order to take advantage of error tracking, and to avoid issues with namespaces that contain `then` function exports.

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

Creating a new `vm.Script` object compiles `code` but does not run it. The compiled `vm.Script` can be run later multiple times. The `code` is not bound to any global object; rather, it is bound before each run, just for that run.

script.cachedDataRejected

- `<boolean>` | `<undefined>`

When `cachedData` is supplied to create the `vm.Script`, this value will be set to either `true` or `false` depending on acceptance of the data by V8. Otherwise the value is `undefined`.

script.createCachedData()

- Returns: `<Buffer>`

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.

The code cache of the `Script` doesn't contain any JavaScript observable states. The code cache is safe to be saved along side the script source and used to construct new `Script` instances multiple times.

Functions in the `Script` source can be marked as lazily compiled and they are not compiled at construction of the `Script`. These functions are going to be compiled when they are invoked the first time. The code cache serializes the metadata that V8 currently knows about the `Script` that it can use to speed up future compilations.

```
const script = new vm.Script(`
function add(a, b) {
  return a + b;
}

const x = add(1, 2);
`);

const cacheWithoutAdd = script.createCachedData();
// In `cacheWithoutAdd` the function `add()` is marked for full compilation
// upon invocation.

script.runInThisContext();
```

```
const cacheWithAdd = script.createCachedData();
// `cacheWithAdd` contains fully compiled function `add()`.
```

script.runInContext(contextifiedObject[, options])

- `contextifiedObject` <Object> A contextified object as returned by the `vm.createContext()` method.
- `options` <Object>
  - `displayErrors` <boolean> 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`.
  - `timeout` <integer> 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.
  - `breakOnSigint` <boolean> If `true`, receiving `SIGINT` (`Ctrl` + `C`) will terminate execution and throw an `Error`. Existing handlers for the event that have been attached via `process.on('SIGINT')` are disabled during script execution, but continue to work after that. **Default:** `false`.
- Returns: <any> the result of the very last statement executed in the script.

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.

```
const vm = require('node: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.

script.runInNewContext([contextObject[, options]])

- `contextObject` <Object> An object that will be contextified. If `undefined`, a new object will be created.
- `options` <Object>
  - `displayErrors` <boolean> 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`.
  - `timeout` <integer> 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.
  - `breakOnSigint` <boolean> If `true`, receiving `SIGINT` (`Ctrl` + `C`) will terminate execution and throw an `Error`. Existing handlers for the event that have been attached via `process.on('SIGINT')` are disabled during script execution, but continue to work after that. **Default:** `false`.
  - `contextName` <string> Human-readable name of the newly created context. **Default:** `'VM Context i'`, where `i` is an ascending numerical index of the created context.

- `contextOrigin` `<string>` Origin corresponding 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:** `''`.
- `contextCodeGeneration` `<Object>`
  - `strings` `<boolean>` If set to false any calls to `eval` or function constructors (`Function`, `GeneratorFunction`, etc) will throw an `EvalError`. **Default:** `true`.
  - `wasm` `<boolean>` If set to false any attempt to compile a WebAssembly module will throw a `WebAssembly.CompileError`. **Default:** `true`.
- `microtaskMode` `<string>` If set to `afterEvaluate`, microtasks (tasks scheduled through `Promise`s and `async function`s) will be run immediately after the script has run. They are included in the `timeout` and `breakOnSigint` scopes in that case.

- Returns: `<any>` the result of the very last statement executed in the script.

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`.

```
const vm = require('node: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' }]
```

script.runInThisContext([options])

- `options` `<Object>`
  - `displayErrors` `<boolean>` 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`.
  - `timeout` `<integer>` 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.
  - `breakOnSigint` `<boolean>` If `true`, receiving `SIGINT` (`Ctrl` + `C`) will terminate execution and throw an `Error`. Existing handlers for the event that have been attached via `process.on('SIGINT')` are disabled during script execution, but continue to work after that. **Default:** `false`.
- Returns: `<any>` the result of the very last statement executed in the script.

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:

```
const vm = require('node: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
```

Class: vm.Module

Stability: 1 - Experimental

This feature is only available with the `--experimental-vm-modules` command flag enabled.

The `vm.Module` class provides a low-level interface for using ECMAScript modules in VM contexts. It is the counterpart of the `vm.Script` class that closely mirrors Module Record s as defined in the ECMAScript specification.

Unlike `vm.Script` however, every `vm.Module` object is bound to a context from its creation. Operations on `vm.Module` objects are intrinsically asynchronous, in contrast with the synchronous nature of `vm.Script` objects. The use of 'async' functions can help with manipulating `vm.Module` objects.

Using a `vm.Module` object requires three distinct steps: creation/parsing, linking, and evaluation. These three steps are illustrated in the following example.

This implementation lies at a lower level than the ECMAScript Module loader . There is also no way to interact with the Loader yet, though support is planned.

CJS ESM

```
import vm from 'node:vm';

const contextifiedObject = vm.createContext({
  secret: 42,
  print: console.log,
});

// Step 1
//
// Create a Module by constructing a new `vm.SourceTextModule` object. This
// parses the provided source text, throwing a `SyntaxError` if anything goes
// wrong. By default, a Module is created in the top context. But here, we
// specify `contextifiedObject` as the context this Module belongs to.
//
// Here, we attempt to obtain the default export from the module "foo", and
// put it into local binding "secret".

const bar = new vm.SourceTextModule(`
  import s from 'foo';
  s;
  print(s);
`, { context: contextifiedObject });

// Step 2
//
// "Link" the imported dependencies of this Module to it.
//
// The provided linking callback (the "linker") accepts two arguments: the
// parent module (`bar` in this case) and the string that is the specifier of
// the imported module. The callback is expected to return a Module that
```

```
// corresponds to the provided specifier, with certain requirements documented
// in `module.link()`.
//
// If linking has not started for the returned Module, the same linker
// callback will be called on the returned Module.
//
// Even top-level Modules without dependencies must be explicitly linked. The
// callback provided would never be called, however.
//
// The link() method returns a Promise that will be resolved when all the
// Promises returned by the linker resolve.
//
// Note: This is a contrived example in that the linker function creates a new
// "foo" module every time it is called. In a full-fledged module system, a
// cache would probably be used to avoid duplicated modules.

async function linker(specifier, referencingModule) {
  if (specifier === 'foo') {
    return new vm.SourceTextModule(`
      // The "secret" variable refers to the global variable we added to
      // "contextifiedObject" when creating the context.
      export default secret;
    `, { context: referencingModule.context });

    // Using `contextifiedObject` instead of `referencingModule.context`
    // here would work as well.
  }
  throw new Error(`Unable to resolve dependency: ${specifier}`);
}
await bar.link(linker);

// Step 3
//
// Evaluate the Module. The evaluate() method returns a promise which will
// resolve after the module has finished evaluating.

// Prints 42.
await bar.evaluate();
```

```
const vm = require('node:vm');

const contextifiedObject = vm.createContext({
  secret: 42,
  print: console.log,
});

(async () => {
  // Step 1
  //
  // Create a Module by constructing a new `vm.SourceTextModule` object. This
  // parses the provided source text, throwing a `SyntaxError` if anything goes
  // wrong. By default, a Module is created in the top context. But here, we
  // specify `contextifiedObject` as the context this Module belongs to.
  //
  // Here, we attempt to obtain the default export from the module "foo", and
  // put it into local binding "secret".
```

```
  const bar = new vm.SourceTextModule(`
    import s from 'foo';
    s;
    print(s);
  `, { context: contextifiedObject });

  // Step 2
  //
  // "Link" the imported dependencies of this Module to it.
  //
  // The provided linking callback (the "linker") accepts two arguments: the
  // parent module (`bar` in this case) and the string that is the specifier of
  // the imported module. The callback is expected to return a Module that
  // corresponds to the provided specifier, with certain requirements documented
  // in `module.link()`.
  //
  // If linking has not started for the returned Module, the same linker
  // callback will be called on the returned Module.
  //
  // Even top-level Modules without dependencies must be explicitly linked. The
  // callback provided would never be called, however.
  //
  // The link() method returns a Promise that will be resolved when all the
  // Promises returned by the linker resolve.
  //
  // Note: This is a contrived example in that the linker function creates a new
  // "foo" module every time it is called. In a full-fledged module system, a
  // cache would probably be used to avoid duplicated modules.

  async function linker(specifier, referencingModule) {
    if (specifier === 'foo') {
      return new vm.SourceTextModule(`
        // The "secret" variable refers to the global variable we added to
        // "contextifiedObject" when creating the context.
        export default secret;
      `, { context: referencingModule.context });

      // Using `contextifiedObject` instead of `referencingModule.context`
      // here would work as well.
    }
    throw new Error(`Unable to resolve dependency: ${specifier}`);
  }
  await bar.link(linker);

  // Step 3
  //
  // Evaluate the Module. The evaluate() method returns a promise which will
  // resolve after the module has finished evaluating.

  // Prints 42.
  await bar.evaluate();
})();
```

module.dependencySpecifiers

- <string[]>

The specifiers of all dependencies of this module. The returned array is frozen to disallow any changes to it.

Corresponds to the `[[RequestedModules]]` field of Cyclic Module Record s in the ECMAScript specification.

module.error

- `<any>`

If the `module.status` is `'errored'`, this property contains the exception thrown by the module during evaluation. If the status is anything else, accessing this property will result in a thrown exception.

The value `undefined` cannot be used for cases where there is not a thrown exception due to possible ambiguity with `throw undefined;`.

Corresponds to the `[[EvaluationError]]` field of Cyclic Module Record s in the ECMAScript specification.

module.evaluate([options])

- `options` `<Object>`
  - `timeout` `<integer>` Specifies the number of milliseconds to evaluate before terminating execution. If execution is interrupted, an `Error` will be thrown. This value must be a strictly positive integer.
  - `breakOnSigint` `<boolean>` If `true`, receiving `SIGINT` (`Ctrl` + `C`) will terminate execution and throw an `Error`. Existing handlers for the event that have been attached via `process.on('SIGINT')` are disabled during script execution, but continue to work after that. **Default:** `false`.
- Returns: `<Promise>` Fulfills with `undefined` upon success.

Evaluate the module.

This must be called after the module has been linked; otherwise it will reject. It could be called also when the module has already been evaluated, in which case it will either do nothing if the initial evaluation ended in success (`module.status` is `'evaluated'`) or it will re-throw the exception that the initial evaluation resulted in (`module.status` is `'errored'`).

This method cannot be called while the module is being evaluated (`module.status` is `'evaluating'`).

Corresponds to the Evaluate() concrete method field of Cyclic Module Record s in the ECMAScript specification.

module.identifier

- `<string>`

The identifier of the current module, as set in the constructor.

module.link(linker)

- `linker` `<Function>`
  - `specifier` `<string>` The specifier of the requested module:

```
import foo from 'foo';
//              ^^^^^ the module specifier
```

  - `referencingModule` `<vm.Module>` The `Module` object `link()` is called on.
  - `extra` `<Object>`
    - `assert` `<Object>` The data from the assertion:

```
import foo from 'foo' assert { name: 'value' };
//                           ^^^^^^^^^^^^^^^^^^ the assertion
```

Per ECMA-262, hosts are expected to ignore assertions that they do not support, as opposed to, for example, triggering an error if an unsupported assertion is present.

    - Returns: `<vm.Module>` | `<Promise>`
- Returns: `<Promise>`

Link module dependencies. This method must be called before evaluation, and can only be called once per module.

The function is expected to return a `Module` object or a `Promise` that eventually resolves to a `Module` object. The returned `Module` must satisfy the following two invariants:

- It must belong to the same context as the parent `Module`.
- Its `status` must not be `'errored'`.

If the returned `Module`'s `status` is `'unlinked'`, this method will be recursively called on the returned `Module` with the same provided `linker` function.

`link()` returns a `Promise` that will either get resolved when all linking instances resolve to a valid `Module`, or rejected if the linker function either throws an exception or returns an invalid `Module`.

The linker function roughly corresponds to the implementation-defined HostResolveImportedModule abstract operation in the ECMAScript specification, with a few key differences:

- The linker function is allowed to be asynchronous while HostResolveImportedModule is synchronous.

The actual HostResolveImportedModule implementation used during module linking is one that returns the modules linked during linking. Since at that point all modules would have been fully linked already, the HostResolveImportedModule implementation is fully synchronous per specification.

Corresponds to the Link() concrete method field of Cyclic Module Record s in the ECMAScript specification.

module.namespace

- `<Object>`

The namespace object of the module. This is only available after linking (`module.link()`) has completed.

Corresponds to the GetModuleNamespace abstract operation in the ECMAScript specification.

module.status

- `<string>`

The current status of the module. Will be one of:

- `'unlinked'`: `module.link()` has not yet been called.
- `'linking'`: `module.link()` has been called, but not all Promises returned by the linker function have been resolved yet.
- `'linked'`: The module has been linked successfully, and all of its dependencies are linked, but `module.evaluate()` has not yet been called.
- `'evaluating'`: The module is being evaluated through a `module.evaluate()` on itself or a parent module.
- `'evaluated'`: The module has been successfully evaluated.
- `'errored'`: The module has been evaluated, but an exception was thrown.

Other than `'errored'`, this status string corresponds to the specification's Cyclic Module Record 's `[[Status]]` field. `'errored'` corresponds to `'evaluated'` in the specification, but with `[[EvaluationError]]` set to a value that is not `undefined`.

Class: vm.SourceTextModule

Stability: 1 - Experimental

This feature is only available with the `--experimental-vm-modules` command flag enabled.

- Extends: `<vm.Module>`

The `vm.SourceTextModule` class provides the Source Text Module Record as defined in the ECMAScript specification.

new vm.SourceTextModule(code[, options])

- `code` `<string>` JavaScript Module code to parse
- `options`
  - `identifier` `<string>` String used in stack traces. **Default:** `'vm:module(i)'` where `i` is a context-specific ascending index.
  - `cachedData` `<Buffer>` | `<TypedArray>` | `<DataView>` Provides an optional `Buffer` or `TypedArray`, or `DataView` with V8's code cache data for the supplied source. The `code` must be the same as the module from which this `cachedData` was created.
  - `context` `<Object>` The contextified object as returned by the `vm.createContext()` method, to compile and evaluate this `Module` in.
  - `lineOffset` `<integer>` Specifies the line number offset that is displayed in stack traces produced by this `Module`. **Default:** `0`.
  - `columnOffset` `<integer>` Specifies the first-line column number offset that is displayed in stack traces produced by this `Module`. **Default:** `0`.
  - `initializeImportMeta` `<Function>` Called during evaluation of this `Module` to initialize the `import.meta`.
    - `meta` `<import.meta>`
    - `module` `<vm.SourceTextModule>`
  - `importModuleDynamically` `<Function>` Called during evaluation of this module when `import()` is called. If this option is not specified, calls to `import()` will reject with `ERR_VM_DYNAMIC_IMPORT_CALLBACK_MISSING`.
    - `specifier` `<string>` specifier passed to `import()`
    - `module` `<vm.Module>`
    - `importAssertions` `<Object>` The `"assert"` value passed to the `optionsExpression` optional parameter, or an empty object if no value was provided.
    - Returns: `<Module Namespace Object>` | `<vm.Module>` Returning a `vm.Module` is recommended in order to take advantage of error tracking, and to avoid issues with namespaces that contain `then` function exports.

Creates a new `SourceTextModule` instance.

Properties assigned to the `import.meta` object that are objects may allow the module to access information outside the specified `context`. Use `vm.runInContext()` to create objects in a specific context.

CJS ESM

```mjs
import vm from 'node:vm';

const contextifiedObject = vm.createContext({ secret: 42 });

const module = new vm.SourceTextModule(
  'Object.getPrototypeOf(import.meta.prop).secret = secret;',
  {
    initializeImportMeta(meta) {
      // Note: this object is created in the top context. As such,
      // Object.getPrototypeOf(import.meta.prop) points to the
      // Object.prototype in the top context rather than that in
      // the contextified object.
      meta.prop = {};
    }
  });
// Since module has no dependencies, the linker function will never be called.
await module.link(() => {});
await module.evaluate();
```

```
// Now, Object.prototype.secret will be equal to 42.
//
// To fix this problem, replace
//     meta.prop = {};
// above with
//     meta.prop = vm.runInContext('{}', contextifiedObject);
```

```
const vm = require('node:vm');
const contextifiedObject = vm.createContext({ secret: 42 });
(async () => {
  const module = new vm.SourceTextModule(
    'Object.getPrototypeOf(import.meta.prop).secret = secret;',
    {
      initializeImportMeta(meta) {
        // Note: this object is created in the top context. As such,
        // Object.getPrototypeOf(import.meta.prop) points to the
        // Object.prototype in the top context rather than that in
        // the contextified object.
        meta.prop = {};
      }
    });
  // Since module has no dependencies, the linker function will never be called.
  await module.link(() => {});
  await module.evaluate();
  // Now, Object.prototype.secret will be equal to 42.
  //
  // To fix this problem, replace
  //     meta.prop = {};
  // above with
  //     meta.prop = vm.runInContext('{}', contextifiedObject);
})();
```

sourceTextModule.createCachedData()

- Returns: <Buffer>

Creates a code cache that can be used with the `SourceTextModule` constructor's `cachedData` option. Returns a `Buffer`. This method may be called any number of times before the module has been evaluated.

The code cache of the `SourceTextModule` doesn't contain any JavaScript observable states. The code cache is safe to be saved along side the script source and used to construct new `SourceTextModule` instances multiple times.

Functions in the `SourceTextModule` source can be marked as lazily compiled and they are not compiled at construction of the `SourceTextModule`. These functions are going to be compiled when they are invoked the first time. The code cache serializes the metadata that V8 currently knows about the `SourceTextModule` that it can use to speed up future compilations.

```
// Create an initial module
const module = new vm.SourceTextModule('const a = 1;');

// Create cached data from this module
const cachedData = module.createCachedData();

// Create a new module using the cached data. The code must be the same.
const module2 = new vm.SourceTextModule('const a = 1;', { cachedData });
```

Class: vm.SyntheticModule

Stability: 1 - Experimental

This feature is only available with the `--experimental-vm-modules` command flag enabled.

- Extends: <vm.Module>

The `vm.SyntheticModule` class provides the Synthetic Module Record as defined in the WebIDL specification. The purpose of synthetic modules is to provide a generic interface for exposing non-JavaScript sources to ECMAScript module graphs.

```
const vm = require('node:vm');

const source = '{ "a": 1 }';
const module = new vm.SyntheticModule(['default'], function() {
  const obj = JSON.parse(source);
  this.setExport('default', obj);
});

// Use `module` in linking...
```

new vm.SyntheticModule(exportNames, evaluateCallback[, options])

- `exportNames` <string[]> Array of names that will be exported from the module.
- `evaluateCallback` <Function> Called when the module is evaluated.
- `options`
  - `identifier` <string> String used in stack traces. **Default:** `'vm:module(i)'` where `i` is a context-specific ascending index.
  - `context` <Object> The contextified object as returned by the `vm.createContext()` method, to compile and evaluate this `Module` in.

Creates a new `SyntheticModule` instance.

Objects assigned to the exports of this instance may allow importers of the module to access information outside the specified `context`. Use `vm.runInContext()` to create objects in a specific context.

syntheticModule.setExport(name, value)

- `name` <string> Name of the export to set.
- `value` <any> The value to set the export to.

This method is used after the module is linked to set the values of exports. If it is called before the module is linked, an ERR_VM_MODULE_STATUS error will be thrown.

CJS ESM

```
import vm from 'node:vm';

const m = new vm.SyntheticModule(['x'], () => {
  m.setExport('x', 1);
});

await m.link(() => {});
await m.evaluate();

assert.strictEqual(m.namespace.x, 1);
```

```
const vm = require('node:vm');
(async () => {
  const m = new vm.SyntheticModule(['x'], () => {
    m.setExport('x', 1);
  });
  await m.link(() => {});
  await m.evaluate();
  assert.strictEqual(m.namespace.x, 1);
})();
```

vm.compileFunction(code[, params[, options]])

- `code` `<string>` The body of the function to compile.
- `params` `<string[]>` An array of strings containing all parameters for the function.
- `options` `<Object>`
  - `filename` `<string>` Specifies the filename used in stack traces produced by this script. **Default:** `''`.
  - `lineOffset` `<number>` Specifies the line number offset that is displayed in stack traces produced by this script. **Default:** `0`.
  - `columnOffset` `<number>` Specifies the first-line column number offset that is displayed in stack traces produced by this script. **Default:** `0`.
  - `cachedData` `<Buffer>` | `<TypedArray>` | `<DataView>` Provides an optional `Buffer` or `TypedArray`, or `DataView` with V8's code cache data for the supplied source.
  - `produceCachedData` `<boolean>` Specifies whether to produce new cache data. **Default:** `false`.
  - `parsingContext` `<Object>` The contextified object in which the said function should be compiled in.
  - `contextExtensions` `<Object[]>` An array containing a collection of context extensions (objects wrapping the current scope) to be applied while compiling. **Default:** `[]`.
  - `importModuleDynamically` `<Function>` Called during evaluation of this module when `import()` is called. If this option is not specified, calls to `import()` will reject with `ERR_VM_DYNAMIC_IMPORT_CALLBACK_MISSING`. This option is part of the experimental modules API, and should not be considered stable.
    - `specifier` `<string>` specifier passed to `import()`
    - `function` `<Function>`
    - `importAssertions` `<Object>` The `"assert"` value passed to the `optionsExpression` optional parameter, or an empty object if no value was provided.
    - Returns: `<Module Namespace Object>` | `<vm.Module>` Returning a `vm.Module` is recommended in order to take advantage of error tracking, and to avoid issues with namespaces that contain `then` function exports.
- Returns: `<Function>`

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`.

vm.createContext([contextObject[, options]])

- `contextObject` `<Object>`
- `options` `<Object>`
  - `name` `<string>` Human-readable name of the newly created context. **Default:** `'VM Context i'`, where `i` is an ascending numerical index of the created context.
  - `origin` `<string>` Origin corresponding 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:** `''`.
  - `codeGeneration` `<Object>`
    - `strings` `<boolean>` If set to false any calls to `eval` or function constructors (`Function`, `GeneratorFunction`, etc) will throw an `EvalError`. **Default:** `true`.

        - `wasm` <boolean> If set to false any attempt to compile a WebAssembly module will throw a `WebAssembly.CompileError`. **Default:** `true`.
    - `microtaskMode` <string> If set to `afterEvaluate`, microtasks (tasks scheduled through `Promise`s and `async function`s) will be run immediately after a script has run through `script.runInContext()`. They are included in the `timeout` and `breakOnSigint` scopes in that case.
- Returns: <Object> contextified object.

If given a `contextObject`, the `vm.createContext()` method will prepare that object so that it can be used in calls to `vm.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 has. Outside of scripts run by the vm module, global variables will remain unchanged.

```
const vm = require('node: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.

vm.isContext(object)

- `object` <Object>
- Returns: <boolean>

Returns `true` if the given `object` object has been contextified using `vm.createContext()`.

vm.measureMemory([options])

Stability: 1 - Experimental

Measure the memory known to V8 and used by all contexts known to the current V8 isolate, or the main context.

- `options` <Object> Optional.
    - `mode` <string> Either `'summary'` or `'detailed'`. In summary mode, only the memory measured for the main context will be returned. In detailed mode, the memory measured for all contexts known to the current V8 isolate will be returned. **Default:** `'summary'`
    - `execution` <string> Either `'default'` or `'eager'`. With default execution, the promise will not resolve until after the next scheduled garbage collection starts, which may take a while (or never if the program exits before the next GC). With eager execution, the GC will be started right away to measure the memory. **Default:** `'default'`

- Returns: `<Promise>` If the memory is successfully measured the promise will resolve with an object containing information about the memory usage.

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.

```
const vm = require('node: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);
  });
```

vm.runInContext(code, contextifiedObject[, options])

- `code` `<string>` The JavaScript code to compile and run.
- `contextifiedObject` `<Object>` The contextified object that will be used as the `global` when the `code` is compiled and run.
- `options` `<Object>` | `<string>`
  - `filename` `<string>` Specifies the filename used in stack traces produced by this script. **Default:** `'evalmachine.<anonymous>'`.

- `lineOffset` `<number>` Specifies the line number offset that is displayed in stack traces produced by this script. **Default:** `0`.
- `columnOffset` `<number>` Specifies the first-line column number offset that is displayed in stack traces produced by this script. **Default:** `0`.
- `displayErrors` `<boolean>` 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`.
- `timeout` `<integer>` 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.
- `breakOnSigint` `<boolean>` If `true`, receiving `SIGINT` (`Ctrl` + `C`) will terminate execution and throw an `Error`. Existing handlers for the event that have been attached via `process.on('SIGINT')` are disabled during script execution, but continue to work after that. **Default:** `false`.
- `cachedData` `<Buffer>` | `<TypedArray>` | `<DataView>` Provides an optional `Buffer` or `TypedArray`, or `DataView` with V8's code cache data for the supplied source.
- `importModuleDynamically` `<Function>` Called during evaluation of this module when `import()` is called. If this option is not specified, calls to `import()` will reject with `ERR_VM_DYNAMIC_IMPORT_CALLBACK_MISSING`. This option is part of the experimental modules API. We do not recommend using it in a production environment.
  - `specifier` `<string>` specifier passed to `import()`
  - `script` `<vm.Script>`
  - `importAssertions` `<Object>` The `"assert"` value passed to the `optionsExpression` optional parameter, or an empty object if no value was provided.
  - Returns: `<Module Namespace Object>` | `<vm.Module>` Returning a `vm.Module` is recommended in order to take advantage of error tracking, and to avoid issues with namespaces that contain `then` function exports.
- Returns: `<any>` the result of the very last statement executed in the script.

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 `vm.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:

```
const vm = require('node: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 }
```

vm.runInNewContext(code[, contextObject[, options]])

- `code` `<string>` The JavaScript code to compile and run.
- `contextObject` `<Object>` An object that will be contextified. If `undefined`, a new object will be created.
- `options` `<Object>` | `<string>`
  - `filename` `<string>` Specifies the filename used in stack traces produced by this script. **Default:** `'evalmachine.<anonymous>'`.
  - `lineOffset` `<number>` Specifies the line number offset that is displayed in stack traces produced by this script. **Default:** `0`.
  - `columnOffset` `<number>` Specifies the first-line column number offset that is displayed in stack traces produced by this script. **Default:** `0`.

- `displayErrors` `<boolean>` 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`.
- `timeout` `<integer>` 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.
- `breakOnSigint` `<boolean>` If `true`, receiving `SIGINT` (`Ctrl` + `C`) will terminate execution and throw an `Error`. Existing handlers for the event that have been attached via `process.on('SIGINT')` are disabled during script execution, but continue to work after that. **Default:** `false`.
- `contextName` `<string>` Human-readable name of the newly created context. **Default:** `'VM Context i'`, where `i` is an ascending numerical index of the created context.
- `contextOrigin` `<string>` Origin corresponding 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:** `''`.
- `contextCodeGeneration` `<Object>`
  - `strings` `<boolean>` If set to false any calls to `eval` or function constructors (`Function`, `GeneratorFunction`, etc) will throw an `EvalError`. **Default:** `true`.
  - `wasm` `<boolean>` If set to false any attempt to compile a WebAssembly module will throw a `WebAssembly.CompileError`. **Default:** `true`.
- `cachedData` `<Buffer>` | `<TypedArray>` | `<DataView>` Provides an optional `Buffer` or `TypedArray`, or `DataView` with V8's code cache data for the supplied source.
- `importModuleDynamically` `<Function>` Called during evaluation of this module when `import()` is called. If this option is not specified, calls to `import()` will reject with `ERR_VM_DYNAMIC_IMPORT_CALLBACK_MISSING`. This option is part of the experimental modules API. We do not recommend using it in a production environment.
  - `specifier` `<string>` specifier passed to `import()`
  - `script` `<vm.Script>`
  - `importAssertions` `<Object>` The `"assert"` value passed to the `optionsExpression` optional parameter, or an empty object if no value was provided.
  - Returns: `<Module Namespace Object>` | `<vm.Module>` Returning a `vm.Module` is recommended in order to take advantage of error tracking, and to avoid issues with namespaces that contain `then` function exports.
- `microtaskMode` `<string>` If set to `afterEvaluate`, microtasks (tasks scheduled through `Promise`s and `async function`s) will be run immediately after the script has run. They are included in the `timeout` and `breakOnSigint` scopes in that case.

- Returns: `<any>` the result of the very last statement executed in the script.

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`.

```
const vm = require('node:vm');

const contextObject = {
  animal: 'cat',
  count: 2
};

vm.runInNewContext('count += 1; name = "kitty"', contextObject);
console.log(contextObject);
// Prints: { animal: 'cat', count: 3, name: 'kitty' }
```

vm.runInThisContext(code[, options])

- `code` `<string>` The JavaScript code to compile and run.
- `options` `<Object>` | `<string>`
  - `filename` `<string>` Specifies the filename used in stack traces produced by this script. **Default:** `'evalmachine.<anonymous>'`.
  - `lineOffset` `<number>` Specifies the line number offset that is displayed in stack traces produced by this script. **Default:** `0`.
  - `columnOffset` `<number>` Specifies the first-line column number offset that is displayed in stack traces produced by this script. **Default:** `0`.
  - `displayErrors` `<boolean>` 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`.
  - `timeout` `<integer>` 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.
  - `breakOnSigint` `<boolean>` If `true`, receiving `SIGINT` (`Ctrl` + `C`) will terminate execution and throw an `Error`. Existing handlers for the event that have been attached via `process.on('SIGINT')` are disabled during script execution, but continue to work after that. **Default:** `false`.
  - `cachedData` `<Buffer>` | `<TypedArray>` | `<DataView>` Provides an optional `Buffer` or `TypedArray`, or `DataView` with V8's code cache data for the supplied source.
  - `importModuleDynamically` `<Function>` Called during evaluation of this module when `import()` is called. If this option is not specified, calls to `import()` will reject with `ERR_VM_DYNAMIC_IMPORT_CALLBACK_MISSING`. This option is part of the experimental modules API. We do not recommend using it in a production environment.
    - `specifier` `<string>` specifier passed to `import()`
    - `script` `<vm.Script>`
    - `importAssertions` `<Object>` The `"assert"` value passed to the `optionsExpression` optional parameter, or an empty object if no value was provided.
    - Returns: `<Module Namespace Object>` | `<vm.Module>` Returning a `vm.Module` is recommended in order to take advantage of error tracking, and to avoid issues with namespaces that contain `then` function exports.
- Returns: `<any>` the result of the very last statement executed in the script.

`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()` function to run the same code:

```
const vm = require('node: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()` *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, e.g. `(0,eval)('code')`.

Example: Running an HTTP server within a VM

When using either `script.runInThisContext()` or `vm.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 `node:http` module the code passed to the context must either call `require('node:http')` on its own, or have a reference to the `node:http` module passed to it. For instance:

```
'use strict';
const vm = require('node:vm');

const code = `
((require) => {
  const http = require('node: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.

What does it mean to "contextify" an object?

All JavaScript executed within Node.js runs within the scope of a "context". According to the V8 Embedder's Guide:

> In V8, a context is an execution environment that allows separate, unrelated, JavaScript applications to run in a single instance of V8. You must explicitly specify the context in which you want any JavaScript code to be run.

When the method `vm.createContext()` is called, the `contextObject` argument (or a newly-created object if `contextObject` is `undefined`) is associated internally with a new instance of a V8 Context. This V8 Context provides the `code` run using the `node:vm` module's methods with an isolated global environment within which it can operate. The process of creating the V8 Context and associating it with the `contextObject` is what this document refers to as "contextifying" the object.

Timeout interactions with asynchronous tasks and Promises

`Promise`s and `async function`s can schedule tasks run by the JavaScript engine asynchronously. By default, these tasks are run after all JavaScript functions on the current stack are done executing. This allows escaping the functionality of the `timeout` and `breakOnSigint` options.

For example, the following code executed by `vm.runInNewContext()` with a timeout of 5 milliseconds schedules an infinite loop to run after a promise resolves. The scheduled loop is never interrupted by the timeout:

```
const vm = require('node:vm');

function loop() {
  console.log('entering loop');
  while (1) console.log(Date.now());
}

vm.runInNewContext(
  'Promise.resolve().then(() => loop());',
```

```
  { loop, console },
  { timeout: 5 }
);
// This is printed *before* 'entering loop' (!)
console.log('done executing');
```

This can be addressed by passing `microtaskMode: 'afterEvaluate'` to the code that creates the `Context`:

```
const vm = require('node:vm');

function loop() {
  while (1) console.log(Date.now());
}

vm.runInNewContext(
  'Promise.resolve().then(() => loop());',
  { loop, console },
  { timeout: 5, microtaskMode: 'afterEvaluate' }
);
```

In this case, the microtask scheduled through `promise.then()` will be run before returning from `vm.runInNewContext()`, and will be interrupted by the `timeout` functionality. This applies only to code running in a `vm.Context`, so e.g. `vm.runInThisContext()` does not take this option.

Promise callbacks are entered into the microtask queue of the context in which they were created. For example, if `() => loop()` is replaced with just `loop` in the above example, then `loop` will be pushed into the global microtask queue, because it is a function from the outer (main) context, and thus will also be able to escape the timeout.

If asynchronous scheduling functions such as `process.nextTick()`, `queueMicrotask()`, `setTimeout()`, `setImmediate()`, etc. are made available inside a `vm.Context`, functions passed to them will be added to global queues, which are shared by all contexts. Therefore, callbacks passed to those functions are not controllable through the timeout either.