



## V8

Source Code: [lib/v8.js](#)

The `node:v8` module exposes APIs that are specific to the version of [V8](#) built into the Node.js binary. It can be accessed using:

```
const v8 = require('node:v8');
```

COPY

### `v8.cachedDataVersionTag()`

- Returns: [<integer>](#)

Returns an integer representing a version tag derived from the V8 version, command-line flags, and detected CPU features. This is useful for determining whether a [vm.Script](#) `cachedData` buffer is compatible with this instance of V8.

```
console.log(v8.cachedDataVersionTag()); // 3947234607
// The value returned by v8.cachedDataVersionTag() is derived from the V8
// version, command-line flags, and detected CPU features. Test that the value
// does indeed update when flags are toggled.
v8.setFlagsFromString('--allow_natives_syntax');
console.log(v8.cachedDataVersionTag()); // 183726201
```

COPY

### `v8.getHeapCodeStatistics()`

- Returns: [<Object>](#)

Get statistics about code and its metadata in the heap, see V8 [GetHeapCodeAndMetadataStatistics](#) API. Returns an object with the following properties:

- `code_and_metadata_size` [<number>](#)
- `bytecode_and_metadata_size` [<number>](#)
- `external_script_source_size` [<number>](#)
- `cpu_profiler_metadata_size` [<number>](#)

```
{
  code_and_metadata_size: 212208,
  bytecode_and_metadata_size: 161368,
  external_script_source_size: 1410794,
  cpu_profiler_metadata_size: 0,
}
```

COPY

### `v8.getHeapSnapshot([options])`

- options [<Object>](#)
  - `exposeInternals` [<boolean>](#) If true, expose internals in the heap snapshot. **Default:** `false`.

- `exposeNumericValues` [<boolean>](#) If true, expose numeric values in artificial fields. **Default:** `false`.
- Returns: [<stream.Readable>](#) A Readable containing the V8 heap snapshot.

Generates a snapshot of the current V8 heap and returns a Readable Stream that may be used to read the JSON serialized representation. This JSON stream format is intended to be used with tools such as Chrome DevTools. The JSON schema is undocumented and specific to the V8 engine. Therefore, the schema may change from one version of V8 to the next.

Creating a heap snapshot requires memory about twice the size of the heap at the time the snapshot is created. This results in the risk of OOM killers terminating the process.

Generating a snapshot is a synchronous operation which blocks the event loop for a duration depending on the heap size.

```
// Print heap snapshot to the console
const v8 = require('node:v8');
const stream = v8.getHeapSnapshot();
stream.pipe(process.stdout);
```

COPY

## v8.getHeapSpaceStatistics()

- Returns: [<Object\[\]>](#)

Returns statistics about the V8 heap spaces, i.e. the segments which make up the V8 heap. Neither the ordering of heap spaces, nor the availability of a heap space can be guaranteed as the statistics are provided via the V8 [GetHeapSpaceStatistics](#) function and may change from one V8 version to the next.

The value returned is an array of objects containing the following properties:

- `space_name` [<string>](#)
- `space_size` [<number>](#)
- `space_used_size` [<number>](#)
- `space_available_size` [<number>](#)
- `physical_space_size` [<number>](#)

```
[
  {
    "space_name": "new_space",
    "space_size": 2063872,
    "space_used_size": 951112,
    "space_available_size": 80824,
    "physical_space_size": 2063872
  },
  {
    "space_name": "old_space",
    "space_size": 3090560,
    "space_used_size": 2493792,
    "space_available_size": 0,
    "physical_space_size": 3090560
  },
  {
    "space_name": "code_space",
    "space_size": 1260160,
    "space_used_size": 644256,
    "space_available_size": 960,
```

```

    "physical_space_size": 1260160
  },
  {
    "space_name": "map_space",
    "space_size": 1094160,
    "space_used_size": 201608,
    "space_available_size": 0,
    "physical_space_size": 1094160
  },
  {
    "space_name": "large_object_space",
    "space_size": 0,
    "space_used_size": 0,
    "space_available_size": 1490980608,
    "physical_space_size": 0
  }
]

```

COPY

## v8.getHeapStatistics()

- Returns: [<Object>](#)

Returns an object with the following properties:

- `total_heap_size` [<number>](#)
- `total_heap_size_executable` [<number>](#)
- `total_physical_size` [<number>](#)
- `total_available_size` [<number>](#)
- `used_heap_size` [<number>](#)
- `heap_size_limit` [<number>](#)
- `malloced_memory` [<number>](#)
- `peak_malloced_memory` [<number>](#)
- `does_zap_garbage` [<number>](#)
- `number_of_native_contexts` [<number>](#)
- `number_of_detached_contexts` [<number>](#)
- `total_global_handles_size` [<number>](#)
- `used_global_handles_size` [<number>](#)
- `external_memory` [<number>](#)

**total\_heap\_size** The value of `total_heap_size` is the number of bytes V8 has allocated for the heap. This can grow if `used_heap` needs more memory.

**total\_heap\_size\_executable** The value of `total_heap_size_executable` is the portion of the heap that can contain executable code, in bytes. This includes memory used by JIT-compiled code and any memory that must be kept executable.

**total\_physical\_size** The value of `total_physical_size` is the actual physical memory used by the V8 heap, in bytes. This is the amount of memory that is committed (or in use) rather than reserved.

**total\_available\_size** The value of `total_available_size` is the number of bytes of memory available to the V8 heap. This value represents how much more memory V8 can use before it exceeds the heap limit.

**used\_heap\_size** The value of `used_heap_size` is number of bytes currently being used by V8's JavaScript objects. This is the actual memory in use and does not include memory that has been allocated but not yet used.

**heap\_size\_limit** The value of `heap_size_limit` is the maximum size of the V8 heap, in bytes (either the default limit, determined by system resources, or the value passed to the `--max_old_space_size` option).

**malloced\_memory** The value of `malloced_memory` is the number of bytes allocated through `malloc` by V8.

**peak\_malloced\_memory** The value of `peak_malloced_memory` is the peak number of bytes allocated through `malloc` by V8 during the lifetime of the process.

**does\_zap\_garbage** is a 0/1 boolean, which signifies whether the `--zap_code_space` option is enabled or not. This makes V8 overwrite heap garbage with a bit pattern. The RSS footprint (resident set size) gets bigger because it continuously touches all heap pages and that makes them less likely to get swapped out by the operating system.

**number\_of\_native\_contexts** The value of `native_context` is the number of the top-level contexts currently active. Increase of this number over time indicates a memory leak.

**number\_of\_detached\_contexts** The value of `detached_context` is the number of contexts that were detached and not yet garbage collected. This number being non-zero indicates a potential memory leak.

**total\_global\_handles\_size** The value of `total_global_handles_size` is the total memory size of V8 global handles.

**used\_global\_handles\_size** The value of `used_global_handles_size` is the used memory size of V8 global handles.

**external\_memory** The value of `external_memory` is the memory size of array buffers and external strings.

```
{
  total_heap_size: 7326976,
  total_heap_size_executable: 4194304,
  total_physical_size: 7326976,
  total_available_size: 1152656,
  used_heap_size: 3476208,
  heap_size_limit: 1535115264,
  malloced_memory: 16384,
  peak_malloced_memory: 1127496,
  does_zap_garbage: 0,
  number_of_native_contexts: 1,
  number_of_detached_contexts: 0,
  total_global_handles_size: 8192,
  used_global_handles_size: 3296,
  external_memory: 318824
}
```

COPY

## v8.queryObjects(ctor[, options])

Stability: 1 .1 - Active development

- **ctor** [<Function>](#) The constructor that can be used to search on the prototype chain in order to filter target objects in the heap.
- **options** [<undefined>](#) | [<Object>](#)
  - **format** [<string>](#) If it's 'count', the count of matched objects is returned. If it's 'summary', an array with summary strings of the matched objects is returned.
- Returns: {number|Array}

This is similar to the [queryObjects\(\) console API](#) provided by the Chromium DevTools console. It can be used to search for objects that have the matching constructor on its prototype chain in the heap after a full garbage collection, which can be useful for memory leak regression tests. To avoid surprising results, users should avoid using this API on constructors whose implementation they don't control, or on constructors that can be invoked by other parties in the application.

To avoid accidental leaks, this API does not return raw references to the objects found. By default, it returns the count of the objects found. If `options.format` is `'summary'`, it returns an array containing brief string representations for each object. The visibility provided in this API is similar to what the heap snapshot provides, while users can save the cost of serialization and parsing and directly filter the target objects during the search.

Only objects created in the current execution context are included in the results.

```
const { queryObjects } = require('node:v8');
class A { foo = 'bar'; }
console.log(queryObjects(A)); // 0
const a = new A();
console.log(queryObjects(A)); // 1
// [ "A { foo: 'bar' }" ]
console.log(queryObjects(A, { format: 'summary' }));

class B extends A { bar = 'qux'; }
const b = new B();
console.log(queryObjects(B)); // 1
// [ "B { foo: 'bar', bar: 'qux' }" ]
console.log(queryObjects(B, { format: 'summary' }));

// Note that, when there are child classes inheriting from a constructor,
// the constructor also shows up in the prototype chain of the child
// classes's prototype, so the child classes's prototype would also be
// included in the result.
console.log(queryObjects(A)); // 3
// [ "B { foo: 'bar', bar: 'qux' }", "A {}", "A { foo: 'bar' }" ]
console.log(queryObjects(A, { format: 'summary' }));
```

---

```
import { queryObjects } from 'node:v8';
class A { foo = 'bar'; }
console.log(queryObjects(A)); // 0
const a = new A();
console.log(queryObjects(A)); // 1
// [ "A { foo: 'bar' }" ]
console.log(queryObjects(A, { format: 'summary' }));

class B extends A { bar = 'qux'; }
const b = new B();
console.log(queryObjects(B)); // 1
// [ "B { foo: 'bar', bar: 'qux' }" ]
console.log(queryObjects(B, { format: 'summary' }));

// Note that, when there are child classes inheriting from a constructor,
// the constructor also shows up in the prototype chain of the child
// classes's prototype, so the child classes's prototype would also be
// included in the result.
```

```
console.log(queryObjects(A)); // 3
// [ "B { foo: 'bar', bar: 'qux' }", 'A {}', "A { foo: 'bar' }" ]
console.log(queryObjects(A, { format: 'summary' }));
```

COPY

## v8.setFlagsFromString(flags)

- flags [<string>](#)

The `v8.setFlagsFromString()` method can be used to programmatically set V8 command-line flags. This method should be used with care. Changing settings after the VM has started may result in unpredictable behavior, including crashes and data loss; or it may simply do nothing.

The V8 options available for a version of Node.js may be determined by running `node --v8-options`.

Usage:

```
// Print GC events to stdout for one minute.
const v8 = require('node:v8');
v8.setFlagsFromString('--trace_gc');
setTimeout(() => { v8.setFlagsFromString('--notrace_gc'); }, 60e3);
```

COPY

## v8.stopCoverage()

The `v8.stopCoverage()` method allows the user to stop the coverage collection started by [NODE\\_V8\\_COVERAGE](#), so that V8 can release the execution count records and optimize code. This can be used in conjunction with [v8.takeCoverage\(\)](#) if the user wants to collect the coverage on demand.

## v8.takeCoverage()

The `v8.takeCoverage()` method allows the user to write the coverage started by [NODE\\_V8\\_COVERAGE](#) to disk on demand. This method can be invoked multiple times during the lifetime of the process. Each time the execution counter will be reset and a new coverage report will be written to the directory specified by [NODE\\_V8\\_COVERAGE](#).

When the process is about to exit, one last coverage will still be written to disk unless [v8.stopCoverage\(\)](#) is invoked before the process exits.

## v8.writeHeapSnapshot([filename[,options]])

- filename [<string>](#) The file path where the V8 heap snapshot is to be saved. If not specified, a file name with the pattern 'Heap-\${yyyymmdd}-\${hhmmss}-\${pid}-\${thread\_id}.heapsnapshot' will be generated, where {pid} will be the PID of the Node.js process, {thread\_id} will be 0 when `writeHeapSnapshot()` is called from the main Node.js thread or the id of a worker thread.
- options [<Object>](#)
  - exposeInternals [<boolean>](#) If true, expose internals in the heap snapshot. **Default:** false.
  - exposeNumericValues [<boolean>](#) If true, expose numeric values in artificial fields. **Default:** false.
- Returns: [<string>](#) The filename where the snapshot was saved.

Generates a snapshot of the current V8 heap and writes it to a JSON file. This file is intended to be used with tools such as Chrome DevTools. The JSON schema is undocumented and specific to the V8 engine, and may change from one version of V8 to the next.

A heap snapshot is specific to a single V8 isolate. When using [worker threads](#), a heap snapshot generated from the main thread will not contain any information about the workers, and vice versa.

Creating a heap snapshot requires memory about twice the size of the heap at the time the snapshot is created. This results in the risk of OOM killers terminating the process.

Generating a snapshot is a synchronous operation which blocks the event loop for a duration depending on the heap size.

```
const { writeHeapSnapshot } = require('node:v8');
const {
  Worker,
  isMainThread,
  parentPort,
} = require('node:worker_threads');

if (isMainThread) {
  const worker = new Worker(__filename);

  worker.once('message', (filename) => {
    console.log(`worker heapdump: ${filename}`);
    // Now get a heapdump for the main thread.
    console.log(`main thread heapdump: ${writeHeapSnapshot()}`);
  });

  // Tell the worker to create a heapdump.
  worker.postMessage('heapdump');
} else {
  parentPort.once('message', (message) => {
    if (message === 'heapdump') {
      // Generate a heapdump for the worker
      // and return the filename to the parent.
      parentPort.postMessage(writeHeapSnapshot());
    }
  });
}
```

COPY

## v8.setHeapSnapshotNearHeapLimit(limit)

Stability: 1 - Experimental

- `limit` [<integer>](#)

The API is a no-op if `--heapsnapshot-near-heap-limit` is already set from the command line or the API is called more than once. `limit` must be a positive integer. See [--heapsnapshot-near-heap-limit](#) for more information.

## Serialization API

The serialization API provides means of serializing JavaScript values in a way that is compatible with the [HTML structured clone algorithm](#).

The format is backward-compatible (i.e. safe to store to disk). Equal JavaScript values may result in different serialized output.

## v8.serialize(value)

- `value` [<any>](#)
- Returns: [<Buffer>](#)

Uses a [DefaultSerializer](#) to serialize `value` into a buffer.

[ERR\\_BUFFER\\_TOO\\_LARGE](#) will be thrown when trying to serialize a huge object which requires buffer larger than [buffer.constants.MAX\\_LENGTH](#).

## v8.deserialize(buffer)

- `buffer` [<Buffer>](#) | [<TypedArray>](#) | [<DataView>](#) A buffer returned by [serialize\(\)](#).

Uses a [DefaultDeserializer](#) with default options to read a JS value from a buffer.

## Class: v8.Serializer

### new Serializer()

Creates a new `Serializer` object.

### serializer.writeHeader()

Writes out a header, which includes the serialization format version.

### serializer.writeValue(value)

- `value` [<any>](#)

Serializes a JavaScript value and adds the serialized representation to the internal buffer.

This throws an error if `value` cannot be serialized.

### serializer.releaseBuffer()

- Returns: [<Buffer>](#)

Returns the stored internal buffer. This serializer should not be used once the buffer is released. Calling this method results in undefined behavior if a previous write has failed.

### serializer.transferArrayBuffer(id, arrayBuffer)

- `id` [<integer>](#) A 32-bit unsigned integer.
- `arrayBuffer` [<ArrayBuffer>](#) An `ArrayBuffer` instance.

Marks an `ArrayBuffer` as having its contents transferred out of band. Pass the corresponding `ArrayBuffer` in the deserializing context to [deserializer.transferArrayBuffer\(\)](#).

### serializer.writeUint32(value)

- `value` [<integer>](#)

Write a raw 32-bit unsigned integer. For use inside of a custom [serializer.writeHostObject\(\)](#).

### serializer.writeUint64(hi, lo)

- `hi` [<integer>](#)
- `lo` [<integer>](#)

Write a raw 64-bit unsigned integer, split into high and low 32-bit parts. For use inside of a custom [serializer.writeHostObject\(\)](#).

### serializer.writeDouble(value)

- `value` [<number>](#)

Write a JS `number` value. For use inside of a custom [serializer.writeHostObject\(\)](#).



## serializer.writeRawBytes(buffer)

- buffer [<Buffer>](#) | [<TypedArray>](#) | [<DataView>](#)

Write raw bytes into the serializer's internal buffer. The deserializer will require a way to compute the length of the buffer. For use inside of a custom [serializer.writeHostObject\(\)](#).

## serializer.\_writeHostObject(object)

- object [<Object>](#)

This method is called to write some kind of host object, i.e. an object created by native C++ bindings. If it is not possible to serialize `object`, a suitable exception should be thrown.

This method is not present on the `Serializer` class itself but can be provided by subclasses.

## serializer.\_getDataCloneError(message)

- message [<string>](#)

This method is called to generate error objects that will be thrown when an object can not be cloned.

This method defaults to the [Error](#) constructor and can be overridden on subclasses.

## serializer.\_getSharedArrayBufferId(sharedArrayBuffer)

- sharedArrayBuffer [<SharedArrayBuffer>](#)

This method is called when the serializer is going to serialize a `SharedArrayBuffer` object. It must return an unsigned 32-bit integer ID for the object, using the same ID if this `SharedArrayBuffer` has already been serialized. When deserializing, this ID will be passed to [deserializer.transferArrayBuffer\(\)](#).

If the object cannot be serialized, an exception should be thrown.

This method is not present on the `Serializer` class itself but can be provided by subclasses.

## serializer.\_setTreatArrayBufferViewsAsHostObjects(flag)

- flag [<boolean>](#) Default: false

Indicate whether to treat `TypedArray` and `DataView` objects as host objects, i.e. pass them to [serializer.writeHostObject\(\)](#).

## Class: v8.Deserializer

### new Deserializer(buffer)

- buffer [<Buffer>](#) | [<TypedArray>](#) | [<DataView>](#) A buffer returned by [serializer.releaseBuffer\(\)](#).

Creates a new `Deserializer` object.

### deserializer.readHeader()

Reads and validates a header (including the format version). May, for example, reject an invalid or unsupported wire format. In that case, an `Error` is thrown.

### deserializer.readValue()

Deserializes a JavaScript value from the buffer and returns it.

### deserializer.transferArrayBuffer(id, arrayBuffer)

- `id` [<integer>](#) A 32-bit unsigned integer.
- `arrayBuffer` [<ArrayBuffer>](#) | [<SharedArrayBuffer>](#) An `ArrayBuffer` instance.

Marks an `ArrayBuffer` as having its contents transferred out of band. Pass the corresponding `ArrayBuffer` in the serializing context to [`serializer.transferArrayBuffer\(\)`](#) (or return the `id` from [`serializer.\_getSharedArrayBufferId\(\)`](#) in the case of `SharedArrayBuffer` s).

### **`deserializer.getWireFormatVersion()`**

- Returns: [<integer>](#)

Reads the underlying wire format version. Likely mostly to be useful to legacy code reading old wire format versions. May not be called before `.readHeader()` .

### **`deserializer.readUint32()`**

- Returns: [<integer>](#)

Read a raw 32-bit unsigned integer and return it. For use inside of a custom [`deserializer.\_readHostObject\(\)`](#) .

### **`deserializer.readUint64()`**

- Returns: [<integer\[\]>](#)

Read a raw 64-bit unsigned integer and return it as an array `[hi, lo]` with two 32-bit unsigned integer entries. For use inside of a custom [`deserializer.\_readHostObject\(\)`](#) .

### **`deserializer.readDouble()`**

- Returns: [<number>](#)

Read a JS `number` value. For use inside of a custom [`deserializer.\_readHostObject\(\)`](#) .

### **`deserializer.readRawBytes(length)`**

- `length` [<integer>](#)
- Returns: [<Buffer>](#)

Read raw bytes from the deserializer's internal buffer. The `length` parameter must correspond to the length of the buffer that was passed to [`serializer.writeRawBytes\(\)`](#) . For use inside of a custom [`deserializer.\_readHostObject\(\)`](#) .

### **`deserializer._readHostObject()`**

This method is called to read some kind of host object, i.e. an object that is created by native C++ bindings. If it is not possible to deserialize the data, a suitable exception should be thrown.

This method is not present on the `Deserializer` class itself but can be provided by subclasses.

## **Class: `v8.DefaultSerializer`**

A subclass of [`Serializer`](#) that serializes `TypedArray` (in particular [`Buffer`](#) ) and `DataView` objects as host objects, and only stores the part of their underlying `ArrayBuffer` s that they are referring to.

## **Class: `v8.DefaultDeserializer`**

A subclass of [`Deserializer`](#) corresponding to the format written by [`DefaultSerializer`](#) .

## **Promise hooks**

The `promiseHooks` interface can be used to track promise lifecycle events. To track *all* async activity, see [async\\_hooks](#) which internally uses this module to produce promise lifecycle events in addition to events for other async resources. For request context management, see [AsyncLocalStorage](#).

```
import { promiseHooks } from 'node:v8';

// There are four lifecycle events produced by promises:

// The `init` event represents the creation of a promise. This could be a
// direct creation such as with `new Promise(...)` or a continuation such
// as `then()` or `catch()`. It also happens whenever an async function is
// called or does an `await`. If a continuation promise is created, the
// `parent` will be the promise it is a continuation from.
function init(promise, parent) {
  console.log('a promise was created', { promise, parent });
}

// The `settled` event happens when a promise receives a resolution or
// rejection value. This may happen synchronously such as when using
// `Promise.resolve()` on non-promise input.
function settled(promise) {
  console.log('a promise resolved or rejected', { promise });
}

// The `before` event runs immediately before a `then()` or `catch()` handler
// runs or an `await` resumes execution.
function before(promise) {
  console.log('a promise is about to call a then handler', { promise });
}

// The `after` event runs immediately after a `then()` handler runs or when
// an `await` begins after resuming from another.
function after(promise) {
  console.log('a promise is done calling a then handler', { promise });
}

// Lifecycle hooks may be started and stopped individually
const stopWatchingInits = promiseHooks.onInit(init);
const stopWatchingSettleds = promiseHooks.onSettled(settled);
const stopWatchingBefores = promiseHooks.onBefore(before);
const stopWatchingAfters = promiseHooks.onAfter(after);

// Or they may be started and stopped in groups
const stopHookSet = promiseHooks.createHook({
  init,
  settled,
  before,
  after,
});

// To stop a hook, call the function returned at its creation.
stopWatchingInits();
stopWatchingSettleds();
```

```
stopWatchingBefore();
stopWatchingAfters();
stopHookSet();
```

COPY

## promiseHooks.onInit(init)

- `init` [<Function>](#) The [init callback](#) to call when a promise is created.
- Returns: [<Function>](#) Call to stop the hook.

The `init` hook must be a plain function. Providing an `async` function will throw as it would produce an infinite microtask loop.

```
import { promiseHooks } from 'node:v8';

const stop = promiseHooks.onInit((promise, parent) => {});
```

---

```
const { promiseHooks } = require('node:v8');

const stop = promiseHooks.onInit((promise, parent) => {});
```

COPY

## promiseHooks.onSettled(settled)

- `settled` [<Function>](#) The [settled callback](#) to call when a promise is resolved or rejected.
- Returns: [<Function>](#) Call to stop the hook.

The `settled` hook must be a plain function. Providing an `async` function will throw as it would produce an infinite microtask loop.

```
import { promiseHooks } from 'node:v8';

const stop = promiseHooks.onSettled((promise) => {});
```

---

```
const { promiseHooks } = require('node:v8');

const stop = promiseHooks.onSettled((promise) => {});
```

COPY

## promiseHooks.onBefore(before)

- `before` [<Function>](#) The [before callback](#) to call before a promise continuation executes.
- Returns: [<Function>](#) Call to stop the hook.

The `before` hook must be a plain function. Providing an `async` function will throw as it would produce an infinite microtask loop.

```
import { promiseHooks } from 'node:v8';

const stop = promiseHooks.onBefore((promise) => {});
```

---

```
const { promiseHooks } = require('node:v8');

const stop = promiseHooks.onBefore((promise) => {});
```

COPY

## promiseHooks.onAfter(after)

- `after` [<Function>](#) The [after callback](#) to call after a promise continuation executes.
- Returns: [<Function>](#) Call to stop the hook.

The `after` hook must be a plain function. Providing an async function will throw as it would produce an infinite microtask loop.

```
import { promiseHooks } from 'node:v8';

const stop = promiseHooks.onAfter((promise) => {});
```

---

```
const { promiseHooks } = require('node:v8');

const stop = promiseHooks.onAfter((promise) => {});
```

COPY

## promiseHooks.createHook(callbacks)

- `callbacks` [<Object>](#) The [Hook Callbacks](#) to register
  - `init` [<Function>](#) The [init callback](#).
  - `before` [<Function>](#) The [before callback](#).
  - `after` [<Function>](#) The [after callback](#).
  - `settled` [<Function>](#) The [settled callback](#).
- Returns: [<Function>](#) Used for disabling hooks

The hook callbacks must be plain functions. Providing async functions will throw as it would produce an infinite microtask loop.

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

The callbacks `init()` / `before()` / `after()` / `settled()` are called for the respective events during a promise's lifetime.

All callbacks are optional. For example, if only promise creation needs to be tracked, then only the `init` callback needs to be passed. The specifics of all functions that can be passed to `callbacks` is in the [Hook Callbacks](#) section.

```
import { promiseHooks } from 'node:v8';

const stopAll = promiseHooks.createHook({
  init(promise, parent) {},
});
```

---

```
const { promiseHooks } = require('node:v8');

const stopAll = promiseHooks.createHook({
```

```
    init(promise, parent) {},
  });
```

COPY

## Hook callbacks

Key events in the lifetime of a promise have been categorized into four areas: creation of a promise, before/after a continuation handler is called or around an `await`, and when the promise resolves or rejects.

While these hooks are similar to those of [async\\_hooks](#) they lack a `destroy` hook. Other types of async resources typically represent sockets or file descriptors which have a distinct "closed" state to express the `destroy` lifecycle event while promises remain usable for as long as code can still reach them. Garbage collection tracking is used to make promises fit into the `async_hooks` event model, however this tracking is very expensive and they may not necessarily ever even be garbage collected.

Because promises are asynchronous resources whose lifecycle is tracked via the promise hooks mechanism, the `init()`, `before()`, `after()`, and `settled()` callbacks *must not* be async functions as they create more promises which would produce an infinite loop.

While this API is used to feed promise events into [async\\_hooks](#), the ordering between the two is undefined. Both APIs are multi-tenant and therefore could produce events in any order relative to each other.

### init(promise, parent)

- `promise` [<Promise>](#) The promise being created.
- `parent` [<Promise>](#) The promise continued from, if applicable.

Called when a promise is constructed. This *does not* mean that corresponding `before` / `after` events will occur, only that the possibility exists. This will happen if a promise is created without ever getting a continuation.

### before(promise)

- `promise` [<Promise>](#)

Called before a promise continuation executes. This can be in the form of `then()`, `catch()`, or `finally()` handlers or an `await` resuming.

The `before` callback will be called 0 to N times. The `before` callback will typically be called 0 times if no continuation was ever made for the promise. The `before` callback may be called many times in the case where many continuations have been made from the same promise.

### after(promise)

- `promise` [<Promise>](#)

Called immediately after a promise continuation executes. This may be after a `then()`, `catch()`, or `finally()` handler or before an `await` after another `await`.

### settled(promise)

- `promise` [<Promise>](#)

Called when the promise receives a resolution or rejection value. This may occur synchronously in the case of `Promise.resolve()` or `Promise.reject()`.

## Startup Snapshot API

Stability: 1 - Experimental

The `v8.startupSnapshot` interface can be used to add serialization and deserialization hooks for custom startup snapshots.

```
$ node --snapshot-blob snapshot.blob --build-snapshot entry.js
# This launches a process with the snapshot
$ node --snapshot-blob snapshot.blob
```

COPY

In the example above, `entry.js` can use methods from the `v8.startupSnapshot` interface to specify how to save information for custom objects in the snapshot during serialization and how the information can be used to synchronize these objects during deserialization of the snapshot. For example, if the `entry.js` contains the following script:

```
'use strict';

const fs = require('node:fs');
const zlib = require('node:zlib');
const path = require('node:path');
const assert = require('node:assert');

const v8 = require('node:v8');

class BookShelf {
  storage = new Map();

  // Reading a series of files from directory and store them into storage.
  constructor(directory, books) {
    for (const book of books) {
      this.storage.set(book, fs.readFileSync(path.join(directory, book)));
    }
  }

  static compressAll(shelf) {
    for (const [ book, content ] of shelf.storage) {
      shelf.storage.set(book, zlib.gzipSync(content));
    }
  }

  static decompressAll(shelf) {
    for (const [ book, content ] of shelf.storage) {
      shelf.storage.set(book, zlib.gunzipSync(content));
    }
  }
}

// __dirname here is where the snapshot script is placed
// during snapshot building time.
const shelf = new BookShelf(__dirname, [
  'book1.en_US.txt',
  'book1.es_ES.txt',
  'book2.zh_CN.txt',
]);

assert(v8.startupSnapshot.isBuildingSnapshot());
// On snapshot serialization, compress the books to reduce size.
v8.startupSnapshot.addSerializeCallback(BookShelf.compressAll, shelf);
// On snapshot deserialization, decompress the books.
```

```
v8.startupSnapshot.addDeserializeCallback(BookShelf.decompressAll, shelf);
v8.startupSnapshot.setDeserializeMainFunction((shelf) => {
  // process.env and process.argv are refreshed during snapshot
  // deserialization.
  const lang = process.env.BOOK_LANG || 'en_US';
  const book = process.argv[1];
  const name = `${book}.${lang}.txt`;
  console.log(shelf.storage.get(name));
}, shelf);
```

COPY

The resulted binary will get print the data deserialized from the snapshot during start up, using the refreshed `process.env` and `process.argv` of the launched process:

```
$ BOOK_LANG=es_ES node --snapshot-blob snapshot.blob book1
# Prints content of book1.es_ES.txt deserialized from the snapshot.
```

COPY

Currently the application deserialized from a user-land snapshot cannot be snapshotted again, so these APIs are only available to applications that are not deserialized from a user-land snapshot.

## v8.startupSnapshot.addSerializeCallback(callback[, data])

- `callback` [<Function>](#) Callback to be invoked before serialization.
- `data` [<any>](#) Optional data that will be passed to the `callback` when it gets called.

Add a callback that will be called when the Node.js instance is about to get serialized into a snapshot and exit. This can be used to release resources that should not or cannot be serialized or to convert user data into a form more suitable for serialization.

Callbacks are run in the order in which they are added.

## v8.startupSnapshot.addDeserializeCallback(callback[, data])

- `callback` [<Function>](#) Callback to be invoked after the snapshot is deserialized.
- `data` [<any>](#) Optional data that will be passed to the `callback` when it gets called.

Add a callback that will be called when the Node.js instance is deserialized from a snapshot. The `callback` and the `data` (if provided) will be serialized into the snapshot, they can be used to re-initialize the state of the application or to re-acquire resources that the application needs when the application is restarted from the snapshot.

Callbacks are run in the order in which they are added.

## v8.startupSnapshot.setDeserializeMainFunction(callback[, data])

- `callback` [<Function>](#) Callback to be invoked as the entry point after the snapshot is deserialized.
- `data` [<any>](#) Optional data that will be passed to the `callback` when it gets called.

This sets the entry point of the Node.js application when it is deserialized from a snapshot. This can be called only once in the snapshot building script. If called, the deserialized application no longer needs an additional entry point script to start up and will simply invoke the callback along with the deserialized data (if provided), otherwise an entry point script still needs to be provided to the deserialized application.

## v8.startupSnapshot.isBuildingSnapshot()

- Returns: [<boolean>](#)

Returns true if the Node.js instance is run to build a snapshot.

## Class: v8.GCProfiler



This API collects GC data in current thread.

## new v8.GCProfiler()

Create a new instance of the `v8.GCProfiler` class.

## profiler.start()

Start collecting GC data.

## profiler.stop()

Stop collecting GC data and return an object. The content of object is as follows.

```
{
  "version": 1,
  "startTime": 1674059033862,
  "statistics": [
    {
      "gcType": "Scavenge",
      "beforeGC": {
        "heapStatistics": {
          "totalHeapSize": 5005312,
          "totalHeapSizeExecutable": 524288,
          "totalPhysicalSize": 5226496,
          "totalAvailableSize": 4341325216,
          "totalGlobalHandlesSize": 8192,
          "usedGlobalHandlesSize": 2112,
          "usedHeapSize": 4883840,
          "heapSizeLimit": 4345298944,
          "mallocedMemory": 254128,
          "externalMemory": 225138,
          "peakMallocedMemory": 181760
        },
        "heapSpaceStatistics": [
          {
            "spaceName": "read_only_space",
            "spaceSize": 0,
            "spaceUsedSize": 0,
            "spaceAvailableSize": 0,
            "physicalSpaceSize": 0
          }
        ]
      },
      "cost": 1574.14,
      "afterGC": {
        "heapStatistics": {
          "totalHeapSize": 6053888,
          "totalHeapSizeExecutable": 524288,
          "totalPhysicalSize": 5500928,
          "totalAvailableSize": 4341101384,
          "totalGlobalHandlesSize": 8192,
          "usedGlobalHandlesSize": 2112,
          "usedHeapSize": 4059096,
```

```

    "heapSizeLimit": 4345298944,
    "mallocedMemory": 254128,
    "externalMemory": 225138,
    "peakMallocedMemory": 181760
  },
  "heapSpaceStatistics": [
    {
      "spaceName": "read_only_space",
      "spaceSize": 0,
      "spaceUsedSize": 0,
      "spaceAvailableSize": 0,
      "physicalSpaceSize": 0
    }
  ]
}
],
"endTime": 1674059036865
}

```

COPY

Here's an example.

```

const { GCProfiler } = require('v8');
const profiler = new GCProfiler();
profiler.start();
setTimeout(() => {
  console.log(profiler.stop());
}, 1000);

```

COPY