/\*\*

\* The `v8` module exposes APIs that are specific to the version of [V8](https://developers.google.com/v8/) built into the Node.js binary. It can be accessed using:

\*

\* ```js

\* const v8 = require('v8');

\* ```

\* @see [source](https://github.com/nodejs/node/blob/v17.0.0/lib/v8.js)

\*/

declare module 'v8' {

import { Readable } from 'node:stream';

interface HeapSpaceInfo {

space\_name: string;

space\_size: number;

space\_used\_size: number;

space\_available\_size: number;

physical\_space\_size: number;

}

// \*\* Signifies if the --zap\_code\_space option is enabled or not. 1 == enabled, 0 == disabled. \*/

type DoesZapCodeSpaceFlag = 0 | 1;

interface HeapInfo {

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: DoesZapCodeSpaceFlag;

number\_of\_native\_contexts: number;

number\_of\_detached\_contexts: number;

}

interface HeapCodeStatistics {

code\_and\_metadata\_size: number;

bytecode\_and\_metadata\_size: number;

external\_script\_source\_size: number;

}

/\*\*

\* 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.

\*

\* ```js

\* 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

\* ```

\* @since v8.0.0

\*/

function cachedDataVersionTag(): number;

/\*\*

\* Returns an object with the following properties:

\*

\* `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.

\*

\* ```js

\* {

\* 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

\* }

\* ```

\* @since v1.0.0

\*/

function getHeapStatistics(): HeapInfo;

/\*\*

\* 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`](https://v8docs.nodesource.com/node-13.2/d5/dda/classv8\_1\_1\_isolate.html#ac673576f24fdc7a33378f8f57e1d13a4) function and may change from one V8 version to the

\* next.

\*

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

\*

\* ```json

\* [

\* {

\* "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

\* }

\* ]

\* ```

\* @since v6.0.0

\*/

function getHeapSpaceStatistics(): HeapSpaceInfo[];

/\*\*

\* 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:

\*

\* ```js

\* // Print GC events to stdout for one minute.

\* const v8 = require('v8');

\* v8.setFlagsFromString('--trace\_gc');

\* setTimeout(() => { v8.setFlagsFromString('--notrace\_gc'); }, 60e3);

\* ```

\* @since v1.0.0

\*/

function setFlagsFromString(flags: string): void;

/\*\*

\* 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.

\*

\* ```js

\* // Print heap snapshot to the console

\* const v8 = require('v8');

\* const stream = v8.getHeapSnapshot();

\* stream.pipe(process.stdout);

\* ```

\* @since v11.13.0

\* @return A Readable Stream containing the V8 heap snapshot

\*/

function getHeapSnapshot(): Readable;

/\*\*

\* 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.

\*

\* ```js

\* const { writeHeapSnapshot } = require('v8');

\* const {

\* Worker,

\* isMainThread,

\* parentPort

\* } = require('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());

\* }

\* });

\* }

\* ```

\* @since v11.13.0

\* @param filename 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.

\* @return The filename where the snapshot was saved.

\*/

function writeHeapSnapshot(filename?: string): string;

/\*\*

\* Returns an object with the following properties:

\*

\* ```js

\* {

\* code\_and\_metadata\_size: 212208,

\* bytecode\_and\_metadata\_size: 161368,

\* external\_script\_source\_size: 1410794

\* }

\* ```

\* @since v12.8.0

\*/

function getHeapCodeStatistics(): HeapCodeStatistics;

/\*\*

\* @since v8.0.0

\*/

class Serializer {

/\*\*

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

\*/

writeHeader(): void;

/\*\*

\* Serializes a JavaScript value and adds the serialized representation to the

\* internal buffer.

\*

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

\*/

writeValue(val: any): boolean;

/\*\*

\* 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.

\*/

releaseBuffer(): Buffer;

/\*\*

\* Marks an `ArrayBuffer` as having its contents transferred out of band.

\* Pass the corresponding `ArrayBuffer` in the deserializing context to `deserializer.transferArrayBuffer()`.

\* @param id A 32-bit unsigned integer.

\* @param arrayBuffer An `ArrayBuffer` instance.

\*/

transferArrayBuffer(id: number, arrayBuffer: ArrayBuffer): void;

/\*\*

\* Write a raw 32-bit unsigned integer.

\* For use inside of a custom `serializer.\_writeHostObject()`.

\*/

writeUint32(value: number): void;

/\*\*

\* Write a raw 64-bit unsigned integer, split into high and low 32-bit parts.

\* For use inside of a custom `serializer.\_writeHostObject()`.

\*/

writeUint64(hi: number, lo: number): void;

/\*\*

\* Write a JS `number` value.

\* For use inside of a custom `serializer.\_writeHostObject()`.

\*/

writeDouble(value: number): void;

/\*\*

\* 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()`.

\*/

writeRawBytes(buffer: NodeJS.TypedArray): void;

}

/\*\*

\* 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.

\* @since v8.0.0

\*/

class DefaultSerializer extends Serializer {}

/\*\*

\* @since v8.0.0

\*/

class Deserializer {

constructor(data: NodeJS.TypedArray);

/\*\*

\* 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.

\*/

readHeader(): boolean;

/\*\*

\* Deserializes a JavaScript value from the buffer and returns it.

\*/

readValue(): any;

/\*\*

\* 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).

\* @param id A 32-bit unsigned integer.

\* @param arrayBuffer An `ArrayBuffer` instance.

\*/

transferArrayBuffer(id: number, arrayBuffer: ArrayBuffer): void;

/\*\*

\* 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()`.

\*/

getWireFormatVersion(): number;

/\*\*

\* Read a raw 32-bit unsigned integer and return it.

\* For use inside of a custom `deserializer.\_readHostObject()`.

\*/

readUint32(): number;

/\*\*

\* 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()`.

\*/

readUint64(): [number, number];

/\*\*

\* Read a JS `number` value.

\* For use inside of a custom `deserializer.\_readHostObject()`.

\*/

readDouble(): number;

/\*\*

\* 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()`.

\*/

readRawBytes(length: number): Buffer;

}

/\*\*

\* A subclass of `Deserializer` corresponding to the format written by `DefaultSerializer`.

\* @since v8.0.0

\*/

class DefaultDeserializer extends Deserializer {}

/\*\*

\* Uses a `DefaultSerializer` to serialize `value` into a buffer.

\* @since v8.0.0

\*/

function serialize(value: any): Buffer;

/\*\*

\* Uses a `DefaultDeserializer` with default options to read a JS value

\* from a buffer.

\* @since v8.0.0

\* @param buffer A buffer returned by {@link serialize}.

\*/

function deserialize(buffer: NodeJS.TypedArray): any;

/\*\*

\* 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 {@link stopCoverage} is invoked before the process exits.

\* @since v15.1.0, v14.18.0, v12.22.0

\*/

function takeCoverage(): void;

/\*\*

\* 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 {@link takeCoverage} if the user wants to collect the coverage on demand.

\* @since v15.1.0, v14.18.0, v12.22.0

\*/

function stopCoverage(): void;

}

declare module 'node:v8' {

export \* from 'v8';

}