Performance measurement APIs

> *Stability: 2 - Stable*

This module provides an implementation of a subset of the W3C Web Performance APIs as well as additional APIs for Node.js-specific performance measurements.

Node.js supports the following Web Performance APIs:

- High Resolution Time
- Performance Timeline
- User Timing

```
const { PerformanceObserver, performance } = require('perf_hooks');

const obs = new PerformanceObserver((items) => {
  console.log(items.getEntries()[0].duration);
  performance.clearMarks();
});
obs.observe({ type: 'measure' });
performance.measure('Start to Now');

performance.mark('A');
doSomeLongRunningProcess(() => {
  performance.measure('A to Now', 'A');

  performance.mark('B');
  performance.measure('A to B', 'A', 'B');
});
```

`perf_hooks.performance`

An object that can be used to collect performance metrics from the current Node.js instance. It is similar to `window.performance` in browsers.

`performance.clearMarks([name])`

- `name` {string}

If `name` is not provided, removes all `PerformanceMark` objects from the Performance Timeline. If `name` is provided, removes only the named mark.

`performance.clearMeasures([name])`

- `name` {string}

If `name` is not provided, removes all `PerformanceMeasure` objects from the Performance Timeline. If `name` is provided, removes only the named mark.

`performance.eventLoopUtilization([utilization1[, utilization2]])`

- `utilization1` {Object} The result of a previous call to `eventLoopUtilization()`.

- `utilization2` {Object} The result of a previous call to `eventLoopUtilization()` prior to `utilization1`.
- Returns {Object}
  - `idle` {number}
  - `active` {number}
  - `utilization` {number}

The `eventLoopUtilization()` method returns an object that contains the cumulative duration of time the event loop has been both idle and active as a high resolution milliseconds timer. The `utilization` value is the calculated Event Loop Utilization (ELU).

If bootstrapping has not yet finished on the main thread the properties have the value of `0`. The ELU is immediately available on Worker threads since bootstrap happens within the event loop.

Both `utilization1` and `utilization2` are optional parameters.

If `utilization1` is passed, then the delta between the current call's `active` and `idle` times, as well as the corresponding `utilization` value are calculated and returned (similar to `process.hrtime()`).

If `utilization1` and `utilization2` are both passed, then the delta is calculated between the two arguments. This is a convenience option because, unlike `process.hrtime()`, calculating the ELU is more complex than a single subtraction.

ELU is similar to CPU utilization, except that it only measures event loop statistics and not CPU usage. It represents the percentage of time the event loop has spent outside the event loop's event provider (e.g. `epoll_wait`). No other CPU idle time is taken into consideration. The following is an example of how a mostly idle process will have a high ELU.

```
'use strict';
const { eventLoopUtilization } = require('perf_hooks').performance;
const { spawnSync } = require('child_process');

setImmediate(() => {
  const elu = eventLoopUtilization();
  spawnSync('sleep', ['5']);
  console.log(eventLoopUtilization(elu).utilization);
});
```

Although the CPU is mostly idle while running this script, the value of `utilization` is `1`. This is because the call to `child_process.spawnSync()` blocks the event loop from proceeding.

Passing in a user-defined object instead of the result of a previous call to `eventLoopUtilization()` will lead to undefined behavior. The return values are not guaranteed to reflect any correct state of the event loop.

`performance.getEntries()`

- Returns: {PerformanceEntry[]}

Returns a list of `PerformanceEntry` objects in chronological order with respect to `performanceEntry.startTime`. If you are only interested in performance entries of certain types or that have certain names, see `performance.getEntriesByType()` and `performance.getEntriesByName()`.

`performance.getEntriesByName(name[, type])`

- `name` {string}
- `type` {string}
- Returns: {PerformanceEntry[]}

Returns a list of `PerformanceEntry` objects in chronological order with respect to `performanceEntry.startTime` whose `performanceEntry.name` is equal to `name`, and optionally, whose `performanceEntry.entryType` is equal to `type`.

`performance.getEntriesByType(type)`

- `type` {string}
- Returns: {PerformanceEntry[]}

Returns a list of `PerformanceEntry` objects in chronological order with respect to `performanceEntry.startTime` whose `performanceEntry.entryType` is equal to `type`.

`performance.mark([name[, options]])`

- `name` {string}
- `options` {Object}
  - `detail` {any} Additional optional detail to include with the mark.
  - `startTime` {number} An optional timestamp to be used as the mark time. **Defaults**: `performance.now()`.

Creates a new `PerformanceMark` entry in the Performance Timeline. A `PerformanceMark` is a subclass of `PerformanceEntry` whose `performanceEntry.entryType` is always `'mark'`, and whose `performanceEntry.duration` is always `0`. Performance marks are used to mark specific significant moments in the Performance Timeline.

The created `PerformanceMark` entry is put in the global Performance Timeline and can be queried with `performance.getEntries`, `performance.getEntriesByName`, and `performance.getEntriesByType`. When the observation is performed, the entries should be cleared from the global Performance Timeline manually with `performance.clearMarks`.

`performance.measure(name[, startMarkOrOptions[, endMark]])`

- `name` {string}
- `startMarkOrOptions` {string|Object} Optional.
  - `detail` {any} Additional optional detail to include with the measure.
  - `duration` {number} Duration between start and end times.
  - `end` {number|string} Timestamp to be used as the end time, or a string identifying a previously recorded mark.
  - `start` {number|string} Timestamp to be used as the start time, or a string identifying a previously recorded mark.
- `endMark` {string} Optional. Must be omitted if `startMarkOrOptions` is an {Object}.

Creates a new `PerformanceMeasure` entry in the Performance Timeline. A `PerformanceMeasure` is a subclass of `PerformanceEntry` whose `performanceEntry.entryType` is always `'measure'`, and whose `performanceEntry.duration` measures the number of milliseconds elapsed since `startMark` and `endMark`.

The `startMark` argument may identify any *existing* `PerformanceMark` in the Performance Timeline, or *may* identify any of the timestamp properties provided by the `PerformanceNodeTiming` class. If the named `startMark` does not exist, an error is thrown.

The optional `endMark` argument must identify any *existing* `PerformanceMark` in the Performance Timeline or any of the timestamp properties provided by the `PerformanceNodeTiming` class. `endMark` will be `performance.now()` if no parameter is passed, otherwise if the named `endMark` does not exist, an error will be thrown.

The created `PerformanceMeasure` entry is put in the global Performance Timeline and can be queried with `performance.getEntries`, `performance.getEntriesByName`, and `performance.getEntriesByType`. When the observation is performed, the entries should be cleared from the global Performance Timeline manually with `performance.clearMeasures`.

`performance.nodeTiming`

- {PerformanceNodeTiming}

This property is an extension by Node.js. It is not available in Web browsers.

An instance of the `PerformanceNodeTiming` class that provides performance metrics for specific Node.js operational milestones.

`performance.now()`

- Returns: {number}

Returns the current high resolution millisecond timestamp, where 0 represents the start of the current `node` process.

`performance.timeOrigin`

- {number}

The `timeOrigin` specifies the high resolution millisecond timestamp at which the current `node` process began, measured in Unix time.

`performance.timerify(fn[, options])`

- `fn` {Function}
- `options` {Object}
  - `histogram` {RecordableHistogram} A histogram object created using `perf_hooks.createHistogram()` that will record runtime durations in nanoseconds.

This property is an extension by Node.js. It is not available in Web browsers.

Wraps a function within a new function that measures the running time of the wrapped function. A `PerformanceObserver` must be subscribed to the `'function'` event type in order for the timing details to be accessed.

```
const {
  performance,
  PerformanceObserver
} = require('perf_hooks');
```

```
function someFunction() {
  console.log('hello world');
}

const wrapped = performance.timerify(someFunction);

const obs = new PerformanceObserver((list) => {
  console.log(list.getEntries()[0].duration);

  performance.clearMarks();
  performance.clearMeasures();
  obs.disconnect();
});
obs.observe({ entryTypes: ['function'] });

// A performance timeline entry will be created
wrapped();
```

If the wrapped function returns a promise, a finally handler will be attached to the promise and the duration will be reported once the finally handler is invoked.

`performance.toJSON()`

An object which is JSON representation of the `performance` object. It is similar to `window.performance.toJSON` in browsers.

Class: `PerformanceEntry`

`performanceEntry.detail`

- {any}

Additional detail specific to the `entryType`.

`performanceEntry.duration`

- {number}

The total number of milliseconds elapsed for this entry. This value will not be meaningful for all Performance Entry types.

`performanceEntry.entryType`

- {string}

The type of the performance entry. It may be one of:

- `'node'` (Node.js only)
- `'mark'` (available on the Web)
- `'measure'` (available on the Web)
- `'gc'` (Node.js only)
- `'function'` (Node.js only)
- `'http2'` (Node.js only)

- `'http'` (Node.js only)

`performanceEntry.flags`

- {number}

This property is an extension by Node.js. It is not available in Web browsers.

When `performanceEntry.entryType` is equal to `'gc'`, the `performance.flags` property contains additional information about garbage collection operation. The value may be one of:

- `perf_hooks.constants.NODE_PERFORMANCE_GC_FLAGS_NO`
- `perf_hooks.constants.NODE_PERFORMANCE_GC_FLAGS_CONSTRUCT_RETAINED`
- `perf_hooks.constants.NODE_PERFORMANCE_GC_FLAGS_FORCED`
- `perf_hooks.constants.NODE_PERFORMANCE_GC_FLAGS_SYNCHRONOUS_PHANTOM_PROCESSING`
- `perf_hooks.constants.NODE_PERFORMANCE_GC_FLAGS_ALL_AVAILABLE_GARBAGE`
- `perf_hooks.constants.NODE_PERFORMANCE_GC_FLAGS_ALL_EXTERNAL_MEMORY`
- `perf_hooks.constants.NODE_PERFORMANCE_GC_FLAGS_SCHEDULE_IDLE`

`performanceEntry.name`

- {string}

The name of the performance entry.

`performanceEntry.kind`

- {number}

This property is an extension by Node.js. It is not available in Web browsers.

When `performanceEntry.entryType` is equal to `'gc'`, the `performance.kind` property identifies the type of garbage collection operation that occurred. The value may be one of:

- `perf_hooks.constants.NODE_PERFORMANCE_GC_MAJOR`
- `perf_hooks.constants.NODE_PERFORMANCE_GC_MINOR`
- `perf_hooks.constants.NODE_PERFORMANCE_GC_INCREMENTAL`
- `perf_hooks.constants.NODE_PERFORMANCE_GC_WEAKCB`

`performanceEntry.startTime`

- {number}

The high resolution millisecond timestamp marking the starting time of the Performance Entry.

Garbage Collection ('gc') Details

When `performanceEntry.type` is equal to `'gc'`, the `performanceEntry.detail` property will be an {Object} with two properties:

- `kind` {number} One of:
  - `perf_hooks.constants.NODE_PERFORMANCE_GC_MAJOR`
  - `perf_hooks.constants.NODE_PERFORMANCE_GC_MINOR`
  - `perf_hooks.constants.NODE_PERFORMANCE_GC_INCREMENTAL`
  - `perf_hooks.constants.NODE_PERFORMANCE_GC_WEAKCB`
- `flags` {number} One of:

- `perf_hooks.constants.NODE_PERFORMANCE_GC_FLAGS_NO`
- `perf_hooks.constants.NODE_PERFORMANCE_GC_FLAGS_CONSTRUCT_RETAINED`
- `perf_hooks.constants.NODE_PERFORMANCE_GC_FLAGS_FORCED`
- `perf_hooks.constants.NODE_PERFORMANCE_GC_FLAGS_SYNCHRONOUS_PHANTOM_PROCESSING`
- `perf_hooks.constants.NODE_PERFORMANCE_GC_FLAGS_ALL_AVAILABLE_GARBAGE`
- `perf_hooks.constants.NODE_PERFORMANCE_GC_FLAGS_ALL_EXTERNAL_MEMORY`
- `perf_hooks.constants.NODE_PERFORMANCE_GC_FLAGS_SCHEDULE_IDLE`

HTTP/2 ('http2') Details

When `performanceEntry.type` is equal to `'http2'`, the `performanceEntry.detail` property will be an {Object} containing additional performance information.

If `performanceEntry.name` is equal to `Http2Stream`, the `detail` will contain the following properties:

- `bytesRead` {number} The number of `DATA` frame bytes received for this `Http2Stream`.
- `bytesWritten` {number} The number of `DATA` frame bytes sent for this `Http2Stream`.
- `id` {number} The identifier of the associated `Http2Stream`
- `timeToFirstByte` {number} The number of milliseconds elapsed between the `PerformanceEntry` `startTime` and the reception of the first `DATA` frame.
- `timeToFirstByteSent` {number} The number of milliseconds elapsed between the `PerformanceEntry` `startTime` and sending of the first `DATA` frame.
- `timeToFirstHeader` {number} The number of milliseconds elapsed between the `PerformanceEntry` `startTime` and the reception of the first header.

If `performanceEntry.name` is equal to `Http2Session`, the `detail` will contain the following properties:

- `bytesRead` {number} The number of bytes received for this `Http2Session`.
- `bytesWritten` {number} The number of bytes sent for this `Http2Session`.
- `framesReceived` {number} The number of HTTP/2 frames received by the `Http2Session`.
- `framesSent` {number} The number of HTTP/2 frames sent by the `Http2Session`.
- `maxConcurrentStreams` {number} The maximum number of streams concurrently open during the lifetime of the `Http2Session`.
- `pingRTT` {number} The number of milliseconds elapsed since the transmission of a `PING` frame and the reception of its acknowledgment. Only present if a `PING` frame has been sent on the `Http2Session`.
- `streamAverageDuration` {number} The average duration (in milliseconds) for all `Http2Stream` instances.
- `streamCount` {number} The number of `Http2Stream` instances processed by the `Http2Session`.
- `type` {string} Either `'server'` or `'client'` to identify the type of `Http2Session`.

Timerify ('function') Details

When `performanceEntry.type` is equal to `'function'`, the `performanceEntry.detail` property will be an {Array} listing the input arguments to the timed function.

Net ('net') Details

When `performanceEntry.type` is equal to `'net'`, the `performanceEntry.detail` property will be an {Object} containing additional information.

If `performanceEntry.name` is equal to `connect`, the `detail` will contain the following properties: `host`, `port`.

DNS ('dns') Details

When `performanceEntry.type` is equal to `'dns'`, the `performanceEntry.detail` property will be an {Object} containing additional information.

If `performanceEntry.name` is equal to `lookup`, the `detail` will contain the following properties: `hostname`, `family`, `hints`, `verbatim`.

If `performanceEntry.name` is equal to `lookupService`, the `detail` will contain the following properties: `host`, `port`.

If `performanceEntry.name` is equal to `queryxxx` or `getHostByAddr`, the `detail` will contain the following properties: `host`, `ttl`.

Class: `PerformanceNodeTiming`

- Extends: {PerformanceEntry}

This property is an extension by Node.js. It is not available in Web browsers.

Provides timing details for Node.js itself. The constructor of this class is not exposed to users.

`performanceNodeTiming.bootstrapComplete`

- {number}

The high resolution millisecond timestamp at which the Node.js process completed bootstrapping. If bootstrapping has not yet finished, the property has the value of -1.

`performanceNodeTiming.environment`

- {number}

The high resolution millisecond timestamp at which the Node.js environment was initialized.

`performanceNodeTiming.idleTime`

- {number}

The high resolution millisecond timestamp of the amount of time the event loop has been idle within the event loop's event provider (e.g. `epoll_wait`). This does not take CPU usage into consideration. If the event loop has not yet started (e.g., in the first tick of the main script), the property has the value of 0.

`performanceNodeTiming.loopExit`

- {number}

The high resolution millisecond timestamp at which the Node.js event loop exited. If the event loop has not yet exited, the property has the value of -1. It can only have a value of not -1 in a handler of the `'exit'` event.

`performanceNodeTiming.loopStart`

- {number}

The high resolution millisecond timestamp at which the Node.js event loop started. If the event loop has not yet started (e.g., in the first tick of the main script), the property has the value of -1.

`performanceNodeTiming.nodeStart`

- {number}

The high resolution millisecond timestamp at which the Node.js process was initialized.

`performanceNodeTiming.v8Start`

- {number}

The high resolution millisecond timestamp at which the V8 platform was initialized.

Class: `perf_hooks.PerformanceObserver`

`new PerformanceObserver(callback)`

- `callback` {Function}
  - `list` {PerformanceObserverEntryList}
  - `observer` {PerformanceObserver}

`PerformanceObserver` objects provide notifications when new `PerformanceEntry` instances have been added to the Performance Timeline.

```
const {
  performance,
  PerformanceObserver
} = require('perf_hooks');

const obs = new PerformanceObserver((list, observer) => {
  console.log(list.getEntries());

  performance.clearMarks();
  performance.clearMeasures();
  observer.disconnect();
});
obs.observe({ entryTypes: ['mark'], buffered: true });

performance.mark('test');
```

Because `PerformanceObserver` instances introduce their own additional performance overhead, instances should not be left subscribed to notifications indefinitely. Users should disconnect observers as soon as they are no longer needed.

The `callback` is invoked when a `PerformanceObserver` is notified about new `PerformanceEntry` instances. The callback receives a `PerformanceObserverEntryList` instance and a reference to the `PerformanceObserver`.

`performanceObserver.disconnect()`

Disconnects the `PerformanceObserver` instance from all notifications.

`performanceObserver.observe(options)`

- `options` {Object}
  - `type` {string} A single {PerformanceEntry} type. Must not be given if `entryTypes` is already specified.
  - `entryTypes` {string[]} An array of strings identifying the types of {PerformanceEntry} instances the observer is interested in. If not provided an error will be thrown.
  - `buffered` {boolean} If true, the observer callback is called with a list global `PerformanceEntry` buffered entries. If false, only `PerformanceEntry`s created after the time point are sent to the observer callback. **Default:** `false`.

Subscribes the {PerformanceObserver} instance to notifications of new {PerformanceEntry} instances identified either by `options.entryTypes` or `options.type`:

```
const {
  performance,
  PerformanceObserver
} = require('perf_hooks');

const obs = new PerformanceObserver((list, observer) => {
  // Called once asynchronously. `list` contains three items.
});
obs.observe({ type: 'mark' });

for (let n = 0; n < 3; n++)
  performance.mark(`test${n}`);
```

Class: `PerformanceObserverEntryList`

The `PerformanceObserverEntryList` class is used to provide access to the `PerformanceEntry` instances passed to a `PerformanceObserver`. The constructor of this class is not exposed to users.

`performanceObserverEntryList.getEntries()`

- Returns: {PerformanceEntry[]}

Returns a list of `PerformanceEntry` objects in chronological order with respect to `performanceEntry.startTime`.

```
const {
  performance,
  PerformanceObserver
} = require('perf_hooks');

const obs = new PerformanceObserver((perfObserverList, observer) => {
  console.log(perfObserverList.getEntries());
  /**
   * [
   *   PerformanceEntry {
   *     name: 'test',
```

```
   *     entryType: 'mark',
   *     startTime: 81.465639,
   *     duration: 0
   *   },
   *   PerformanceEntry {
   *     name: 'meow',
   *     entryType: 'mark',
   *     startTime: 81.860064,
   *     duration: 0
   *   }
   * ]
   */

  performance.clearMarks();
  performance.clearMeasures();
  observer.disconnect();
});
obs.observe({ type: 'mark' });

performance.mark('test');
performance.mark('meow');
```

`performanceObserverEntryList.getEntriesByName(name[, type])`

- `name` {string}
- `type` {string}
- Returns: {PerformanceEntry[]}

Returns a list of `PerformanceEntry` objects in chronological order with respect to `performanceEntry.startTime` whose `performanceEntry.name` is equal to `name`, and optionally, whose `performanceEntry.entryType` is equal to `type`.

```
const {
  performance,
  PerformanceObserver
} = require('perf_hooks');

const obs = new PerformanceObserver((perfObserverList, observer) => {
  console.log(perfObserverList.getEntriesByName('meow'));
  /**
   * [
   *   PerformanceEntry {
   *     name: 'meow',
   *     entryType: 'mark',
   *     startTime: 98.545991,
   *     duration: 0
   *   }
   * ]
   */
  console.log(perfObserverList.getEntriesByName('nope')); // []
```

```
  console.log(perfObserverList.getEntriesByName('test', 'mark'));
  /**
   * [
   *   PerformanceEntry {
   *     name: 'test',
   *     entryType: 'mark',
   *     startTime: 63.518931,
   *     duration: 0
   *   }
   * ]
   */
  console.log(perfObserverList.getEntriesByName('test', 'measure')); // []

  performance.clearMarks();
  performance.clearMeasures();
  observer.disconnect();
});
obs.observe({ entryTypes: ['mark', 'measure'] });

performance.mark('test');
performance.mark('meow');
```

`performanceObserverEntryList.getEntriesByType(type)`

- `type` {string}
- Returns: {PerformanceEntry[]}

Returns a list of `PerformanceEntry` objects in chronological order with respect to `performanceEntry.startTime` whose `performanceEntry.entryType` is equal to `type`.

```
const {
  performance,
  PerformanceObserver
} = require('perf_hooks');

const obs = new PerformanceObserver((perfObserverList, observer) => {
  console.log(perfObserverList.getEntriesByType('mark'));
  /**
   * [
   *   PerformanceEntry {
   *     name: 'test',
   *     entryType: 'mark',
   *     startTime: 55.897834,
   *     duration: 0
   *   },
   *   PerformanceEntry {
   *     name: 'meow',
   *     entryType: 'mark',
   *     startTime: 56.350146,
   *     duration: 0
   *   }
   * ]
```

```
    */
  performance.clearMarks();
  performance.clearMeasures();
  observer.disconnect();
});
obs.observe({ type: 'mark' });

performance.mark('test');
performance.mark('meow');
```

`perf_hooks.createHistogram([options])`

- `options` {Object}
  - `lowest` {number|bigint} The lowest discernible value. Must be an integer value greater than 0. **Default:** `1`.
  - `highest` {number|bigint} The highest recordable value. Must be an integer value that is equal to or greater than two times `lowest`. **Default:** `Number.MAX_SAFE_INTEGER`.
  - `figures` {number} The number of accuracy digits. Must be a number between `1` and `5`. **Default:** `3`.
- Returns {RecordableHistogram}

Returns a {RecordableHistogram}.

`perf_hooks.monitorEventLoopDelay([options])`

- `options` {Object}
  - `resolution` {number} The sampling rate in milliseconds. Must be greater than zero. **Default:** `10`.
- Returns: {IntervalHistogram}

This property is an extension by Node.js. It is not available in Web browsers.

Creates an `IntervalHistogram` object that samples and reports the event loop delay over time. The delays will be reported in nanoseconds.

Using a timer to detect approximate event loop delay works because the execution of timers is tied specifically to the lifecycle of the libuv event loop. That is, a delay in the loop will cause a delay in the execution of the timer, and those delays are specifically what this API is intended to detect.

```
const { monitorEventLoopDelay } = require('perf_hooks');
const h = monitorEventLoopDelay({ resolution: 20 });
h.enable();
// Do something.
h.disable();
console.log(h.min);
console.log(h.max);
console.log(h.mean);
console.log(h.stddev);
console.log(h.percentiles);
```

```
console.log(h.percentile(50));
console.log(h.percentile(99));
```

Class: `Histogram`

`histogram.count`

- {number}

The number of samples recorded by the histogram.

`histogram.countBigInt`

- {bigint}

The number of samples recorded by the histogram.

`histogram.exceeds`

- {number}

The number of times the event loop delay exceeded the maximum 1 hour event loop delay threshold.

`histogram.exceedsBigInt`

- {bigint}

The number of times the event loop delay exceeded the maximum 1 hour event loop delay threshold.

`histogram.max`

- {number}

The maximum recorded event loop delay.

`histogram.maxBigInt`

- {bigint}

The maximum recorded event loop delay.

`histogram.mean`

- {number}

The mean of the recorded event loop delays.

`histogram.min`

- {number}

The minimum recorded event loop delay.

`histogram.minBigInt`

- {bigint}

The minimum recorded event loop delay.

`histogram.percentile(percentile)`

- `percentile` {number} A percentile value in the range (0, 100].
- Returns: {number}

Returns the value at the given percentile.

`histogram.percentileBigInt(percentile)`

- `percentile` {number} A percentile value in the range (0, 100].
- Returns: {bigint}

Returns the value at the given percentile.

`histogram.percentiles`

- {Map}

Returns a `Map` object detailing the accumulated percentile distribution.

`histogram.percentilesBigInt`

- {Map}

Returns a `Map` object detailing the accumulated percentile distribution.

`histogram.reset()`

Resets the collected histogram data.

`histogram.stddev`

- {number}

The standard deviation of the recorded event loop delays.

Class: `IntervalHistogram extends Histogram`

A `Histogram` that is periodically updated on a given interval.

`histogram.disable()`

- Returns: {boolean}

Disables the update interval timer. Returns `true` if the timer was stopped, `false` if it was already stopped.

`histogram.enable()`

- Returns: {boolean}

Enables the update interval timer. Returns `true` if the timer was started, `false` if it was already started.

Cloning an `IntervalHistogram`

{IntervalHistogram} instances can be cloned via {MessagePort}. On the receiving end, the histogram is cloned as a plain {Histogram} object that does not implement the `enable()` and `disable()` methods.

Class: `RecordableHistogram extends Histogram`

`histogram.add(other)`

- `other` {RecordableHistogram}

Adds the values from `other` to this histogram.

`histogram.record(val)`

- `val` {number|bigint} The amount to record in the histogram.

`histogram.recordDelta()`

Calculates the amount of time (in nanoseconds) that has passed since the previous call to `recordDelta()` and records that amount in the histogram.

Examples

Measuring the duration of async operations

The following example uses the Async Hooks and Performance APIs to measure the actual duration of a Timeout operation (including the amount of time it took to execute the callback).

```
'use strict';
const async_hooks = require('async_hooks');
const {
  performance,
  PerformanceObserver
} = require('perf_hooks');

const set = new Set();
const hook = async_hooks.createHook({
  init(id, type) {
    if (type === 'Timeout') {
      performance.mark(`Timeout-${id}-Init`);
      set.add(id);
    }
  },
  destroy(id) {
    if (set.has(id)) {
      set.delete(id);
      performance.mark(`Timeout-${id}-Destroy`);
      performance.measure(`Timeout-${id}`,
                          `Timeout-${id}-Init`,
                          `Timeout-${id}-Destroy`);
    }
  }
});
hook.enable();

const obs = new PerformanceObserver((list, observer) => {
```

```
  console.log(list.getEntries()[0]);
  performance.clearMarks();
  performance.clearMeasures();
  observer.disconnect();
});
obs.observe({ entryTypes: ['measure'], buffered: true });

setTimeout(() => {}, 1000);
```

Measuring how long it takes to load dependencies

The following example measures the duration of `require()` operations to load dependencies:

```
'use strict';
const {
  performance,
  PerformanceObserver
} = require('perf_hooks');
const mod = require('module');

// Monkey patch the require function
mod.Module.prototype.require =
  performance.timerify(mod.Module.prototype.require);
require = performance.timerify(require);

// Activate the observer
const obs = new PerformanceObserver((list) => {
  const entries = list.getEntries();
  entries.forEach((entry) => {
    console.log(`require('${entry[0]}')`, entry.duration);
  });
  performance.clearMarks();
  performance.clearMeasures();
  obs.disconnect();
});
obs.observe({ entryTypes: ['function'], buffered: true });

require('some-module');
```

Measuring how long one HTTP round-trip takes

The following example is used to trace the time spent by HTTP client (`OutgoingMessage`) and HTTP request (`IncomingMessage`). For HTTP client, it means the time interval between starting the request and receiving the response, and for HTTP request, it means the time interval between receiving the request and sending the response:

```
'use strict';
const { PerformanceObserver } = require('perf_hooks');
const http = require('http');

const obs = new PerformanceObserver((items) => {
```

```
  items.getEntries().forEach((item) => {
    console.log(item);
  });
});

obs.observe({ entryTypes: ['http'] });

const PORT = 8080;

http.createServer((req, res) => {
  res.end('ok');
}).listen(PORT, () => {
  http.get(`http://127.0.0.1:${PORT}`);
});
```

Measuring how long the `net.connect` (only for TCP) takes when the connection is successful

```
'use strict';
const { PerformanceObserver } = require('perf_hooks');
const net = require('net');
const obs = new PerformanceObserver((items) => {
  items.getEntries().forEach((item) => {
    console.log(item);
  });
});
obs.observe({ entryTypes: ['net'] });
const PORT = 8080;
net.createServer((socket) => {
  socket.destroy();
}).listen(PORT, () => {
  net.connect(PORT);
});
```

Measuring how long the DNS takes when the request is successful

```
'use strict';
const { PerformanceObserver } = require('perf_hooks');
const dns = require('dns');
const obs = new PerformanceObserver((items) => {
  items.getEntries().forEach((item) => {
    console.log(item);
  });
});
obs.observe({ entryTypes: ['dns'] });
dns.lookup('localhost', () => {});
dns.promises.resolve('localhost');
```