Timers

Stability: 2 - Stable

Source Code: lib/timers.js

The `timer` module exposes a global API for scheduling functions to be called at some future period of time. Because the timer functions are globals, there is no need to call `require('node:timers')` to use the API.

The timer functions within Node.js implement a similar API as the timers API provided by Web Browsers but use a different internal implementation that is built around the Node.js Event Loop .

Class: Immediate

This object is created internally and is returned from `setImmediate()` . It can be passed to `clearImmediate()` in order to cancel the scheduled actions.

By default, when an immediate is scheduled, the Node.js event loop will continue running as long as the immediate is active. The `Immediate` object returned by `setImmediate()` exports both `immediate.ref()` and `immediate.unref()` functions that can be used to control this default behavior.

immediate.hasRef()

- Returns: `<boolean>`

If true, the `Immediate` object will keep the Node.js event loop active.

immediate.ref()

- Returns: `<Immediate>` a reference to `immediate`

When called, requests that the Node.js event loop *not* exit so long as the `Immediate` is active. Calling `immediate.ref()` multiple times will have no effect.

By default, all `Immediate` objects are "ref'ed", making it normally unnecessary to call `immediate.ref()` unless `immediate.unref()` had been called previously.

immediate.unref()

- Returns: `<Immediate>` a reference to `immediate`

When called, the active `Immediate` object will not require the Node.js event loop to remain active. If there is no other activity keeping the event loop running, the process may exit before the `Immediate` object's callback is invoked. Calling `immediate.unref()` multiple times will have no effect.

immediate[Symbol.dispose]()

Stability: 1 - Experimental

Cancels the immediate. This is similar to calling `clearImmediate()`.

Class: Timeout

This object is created internally and is returned from `setTimeout()` and `setInterval()`. It can be passed to either `clearTimeout()` or `clearInterval()` in order to cancel the scheduled actions.

By default, when a timer is scheduled using either `setTimeout()` or `setInterval()`, the Node.js event loop will continue running as long as the timer is active. Each of the `Timeout` objects returned by these functions export both `timeout.ref()` and `timeout.unref()` functions that can be used to control this default behavior.

timeout.close()

> Stability: 3 - Legacy: Use `clearTimeout()` instead.

- Returns: `<Timeout>` a reference to `timeout`

Cancels the timeout.

timeout.hasRef()

- Returns: `<boolean>`

If true, the `Timeout` object will keep the Node.js event loop active.

timeout.ref()

- Returns: `<Timeout>` a reference to `timeout`

When called, requests that the Node.js event loop *not* exit so long as the `Timeout` is active. Calling `timeout.ref()` multiple times will have no effect.

By default, all `Timeout` objects are "ref'ed", making it normally unnecessary to call `timeout.ref()` unless `timeout.unref()` had been called previously.

timeout.refresh()

- Returns: `<Timeout>` a reference to `timeout`

Sets the timer's start time to the current time, and reschedules the timer to call its callback at the previously specified duration adjusted to the current time. This is useful for refreshing a timer without allocating a new JavaScript object.

Using this on a timer that has already called its callback will reactivate the timer.

timeout.unref()

- Returns: `<Timeout>` a reference to `timeout`

When called, the active `Timeout` object will not require the Node.js event loop to remain active. If there is no other activity keeping the event loop running, the process may exit before the `Timeout` object's callback is invoked. Calling `timeout.unref()` multiple times will have no effect.

timeout[Symbol.toPrimitive]()

- Returns: `<integer>` a number that can be used to reference this `timeout`

Coerce a `Timeout` to a primitive. The primitive can be used to clear the `Timeout`. The primitive can only be used in the same thread where the timeout was created. Therefore, to use it across `worker_threads` it must first be passed to the correct thread. This allows enhanced compatibility with browser `setTimeout()` and `setInterval()` implementations.

timeout[Symbol.dispose]()

Stability: 1 - Experimental

Cancels the timeout.

Scheduling timers

A timer in Node.js is an internal construct that calls a given function after a certain period of time. When a timer's function is called varies depending on which method was used to create the timer and what other work the Node.js event loop is doing.

setImmediate(callback[, ...args])

- `callback` <Function> The function to call at the end of this turn of the Node.js Event Loop
- `...args` <any> Optional arguments to pass when the `callback` is called.
- Returns: <Immediate> for use with `clearImmediate()`

Schedules the "immediate" execution of the `callback` after I/O events' callbacks.

When multiple calls to `setImmediate()` are made, the `callback` functions are queued for execution in the order in which they are created. The entire callback queue is processed every event loop iteration. If an immediate timer is queued from inside an executing callback, that timer will not be triggered until the next event loop iteration.

If `callback` is not a function, a `TypeError` will be thrown.

This method has a custom variant for promises that is available using `timersPromises.setImmediate()`.

setInterval(callback[, delay[, ...args]])

- `callback` <Function> The function to call when the timer elapses.
- `delay` <number> The number of milliseconds to wait before calling the `callback`. **Default:** `1`.
- `...args` <any> Optional arguments to pass when the `callback` is called.
- Returns: <Timeout> for use with `clearInterval()`

Schedules repeated execution of `callback` every `delay` milliseconds.

When `delay` is larger than `2147483647` or less than `1`, the `delay` will be set to `1`. Non-integer delays are truncated to an integer.

If `callback` is not a function, a `TypeError` will be thrown.

This method has a custom variant for promises that is available using `timersPromises.setInterval()`.

setTimeout(callback[, delay[, ...args]])

- `callback` <Function> The function to call when the timer elapses.
- `delay` <number> The number of milliseconds to wait before calling the `callback`. **Default:** `1`.
- `...args` <any> Optional arguments to pass when the `callback` is called.
- Returns: <Timeout> for use with `clearTimeout()`

Schedules execution of a one-time `callback` after `delay` milliseconds.

The `callback` will likely not be invoked in precisely `delay` milliseconds. Node.js makes no guarantees about the exact timing of when callbacks will fire, nor of their ordering. The callback will be called as close as possible to the time specified.

When `delay` is larger than `2147483647` or less than `1`, the `delay` will be set to `1`. Non-integer delays are truncated to an integer.

If `callback` is not a function, a `TypeError` will be thrown.

This method has a custom variant for promises that is available using `timersPromises.setTimeout()`.

Cancelling timers

The `setImmediate()`, `setInterval()`, and `setTimeout()` methods each return objects that represent the scheduled timers. These can be used to cancel the timer and prevent it from triggering.

For the promisified variants of `setImmediate()` and `setTimeout()`, an `AbortController` may be used to cancel the timer. When canceled, the returned Promises will be rejected with an `'AbortError'`.

For `setImmediate()`:

```
const { setImmediate: setImmediatePromise } = require('node:timers/promises');

const ac = new AbortController();
const signal = ac.signal;

setImmediatePromise('foobar', { signal })
  .then(console.log)
  .catch((err) => {
    if (err.name === 'AbortError')
      console.error('The immediate was aborted');
  });

ac.abort();
```

COPY

For `setTimeout()`:

```
const { setTimeout: setTimeoutPromise } = require('node:timers/promises');

const ac = new AbortController();
const signal = ac.signal;

setTimeoutPromise(1000, 'foobar', { signal })
  .then(console.log)
  .catch((err) => {
    if (err.name === 'AbortError')
      console.error('The timeout was aborted');
  });

ac.abort();
```

COPY

clearImmediate(immediate)

- `immediate` `<Immediate>` An `Immediate` object as returned by `setImmediate()`.

Cancels an `Immediate` object created by `setImmediate()`.

clearInterval(timeout)

- `timeout` <Timeout> | <string> | <number> A `Timeout` object as returned by `setInterval()` or the primitive of the `Timeout` object as a string or a number.

Cancels a `Timeout` object created by `setInterval()`.

clearTimeout(timeout)

- `timeout` <Timeout> | <string> | <number> A `Timeout` object as returned by `setTimeout()` or the primitive of the `Timeout` object as a string or a number.

Cancels a `Timeout` object created by `setTimeout()`.

Timers Promises API

The `timers/promises` API provides an alternative set of timer functions that return `Promise` objects. The API is accessible via `require('node:timers/promises')`.

```
import {
  setTimeout,
  setImmediate,
  setInterval,
} from 'node:timers/promises';
```

```
const {
  setTimeout,
  setImmediate,
  setInterval,
} = require('node:timers/promises');
```

COPY

timersPromises.setTimeout([delay[, value[, options]]])

- `delay` <number> The number of milliseconds to wait before fulfilling the promise. **Default:** `1`.
- `value` <any> A value with which the promise is fulfilled.
- `options` <Object>
  - `ref` <boolean> Set to `false` to indicate that the scheduled `Timeout` should not require the Node.js event loop to remain active. **Default:** `true`.
  - `signal` <AbortSignal> An optional `AbortSignal` that can be used to cancel the scheduled `Timeout`.

```
import {
  setTimeout,
} from 'node:timers/promises';

const res = await setTimeout(100, 'result');

console.log(res);  // Prints 'result'
```

```
const {
  setTimeout,
} = require('node:timers/promises');
```

```
setTimeout(100, 'result').then((res) => {
  console.log(res);  // Prints 'result'
});
```

COPY

timersPromises.setImmediate([value[, options]])

- `value` <any> A value with which the promise is fulfilled.
- `options` <Object>
  - `ref` <boolean> Set to `false` to indicate that the scheduled `Immediate` should not require the Node.js event loop to remain active. **Default:** `true`.
  - `signal` <AbortSignal> An optional `AbortSignal` that can be used to cancel the scheduled `Immediate`.

```
import {
  setImmediate,
} from 'node:timers/promises';

const res = await setImmediate('result');

console.log(res);  // Prints 'result'
```

```
const {
  setImmediate,
} = require('node:timers/promises');

setImmediate('result').then((res) => {
  console.log(res);  // Prints 'result'
});
```

COPY

timersPromises.setInterval([delay[, value[, options]]])

Returns an async iterator that generates values in an interval of `delay` ms. If `ref` is `true`, you need to call `next()` of async iterator explicitly or implicitly to keep the event loop alive.

- `delay` <number> The number of milliseconds to wait between iterations. **Default:** `1`.
- `value` <any> A value with which the iterator returns.
- `options` <Object>
  - `ref` <boolean> Set to `false` to indicate that the scheduled `Timeout` between iterations should not require the Node.js event loop to remain active. **Default:** `true`.
  - `signal` <AbortSignal> An optional `AbortSignal` that can be used to cancel the scheduled `Timeout` between operations.

```
import {
  setInterval,
} from 'node:timers/promises';

const interval = 100;
for await (const startTime of setInterval(interval, Date.now())) {
```

```
  const now = Date.now();
  console.log(now);
  if ((now - startTime) > 1000)
    break;
}
console.log(Date.now());
```

```
const {
  setInterval,
} = require('node:timers/promises');
const interval = 100;

(async function() {
  for await (const startTime of setInterval(interval, Date.now())) {
    const now = Date.now();
    console.log(now);
    if ((now - startTime) > 1000)
      break;
  }
  console.log(Date.now());
})();
```

COPY

timersPromises.scheduler.wait(delay[, options])

Stability: 1 - Experimental

- `delay` <number> The number of milliseconds to wait before resolving the promise.
- `options` <Object>
  - `ref` <boolean> Set to `false` to indicate that the scheduled `Timeout` should not require the Node.js event loop to remain active. **Default:** `true`.
  - `signal` <AbortSignal> An optional `AbortSignal` that can be used to cancel waiting.
- Returns: <Promise>

An experimental API defined by the Scheduling APIs draft specification being developed as a standard Web Platform API.

Calling `timersPromises.scheduler.wait(delay, options)` is equivalent to calling `timersPromises.setTimeout(delay, undefined, options)`.

```
import { scheduler } from 'node:timers/promises';

await scheduler.wait(1000); // Wait one second before continuing
```

COPY

timersPromises.scheduler.yield()

Stability: 1 - Experimental

- Returns: `<Promise>`

An experimental API defined by the Scheduling APIs draft specification being developed as a standard Web Platform API.

Calling `timersPromises.scheduler.yield()` is equivalent to calling `timersPromises.setImmediate()` with no arguments.